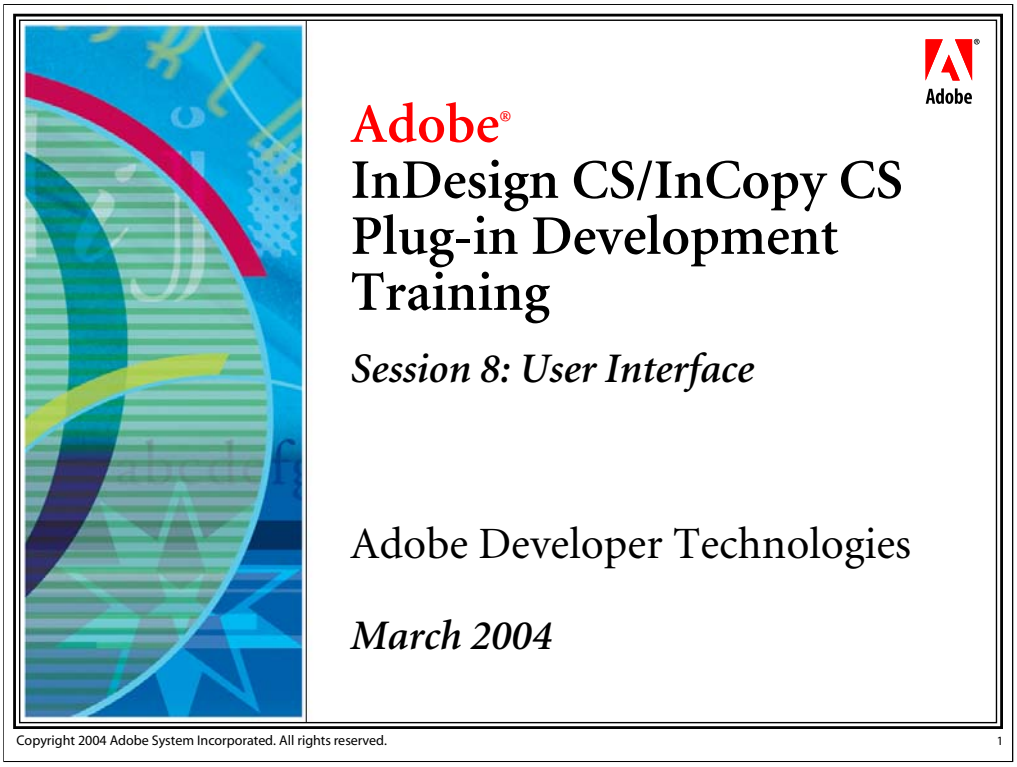
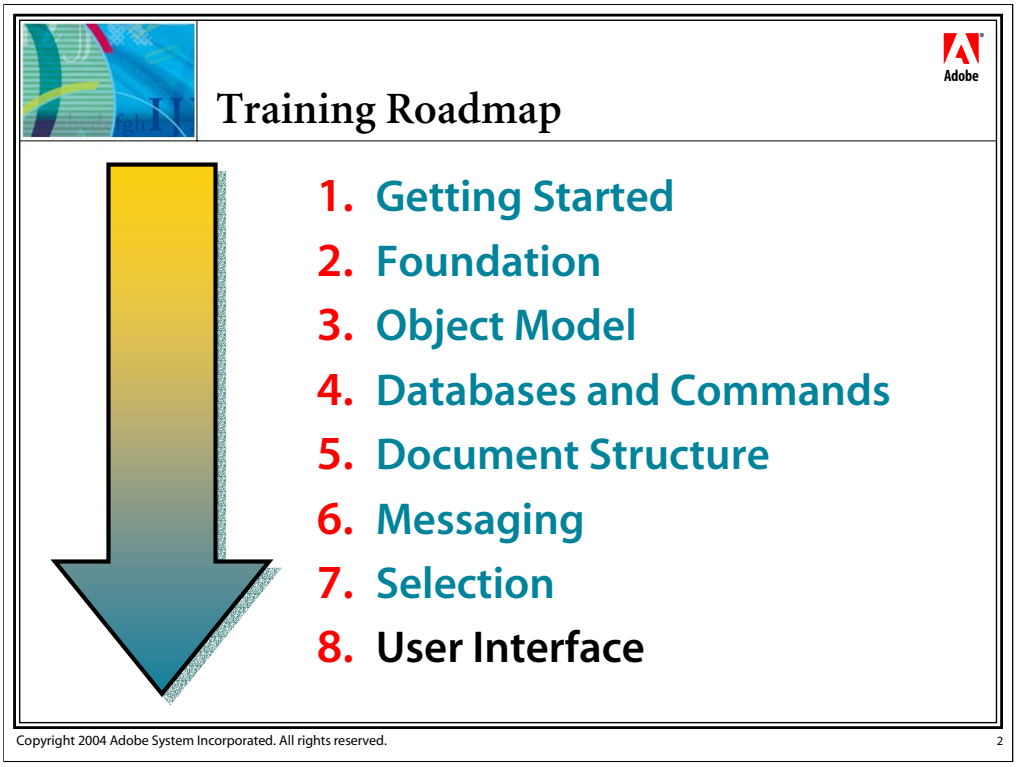


Welcome to the eighth installment of the InDesign CS/InCopy CS Plug-in Development training sessions.

Copyright 2004 Adobe Systems Incorporated. All rights reserved.

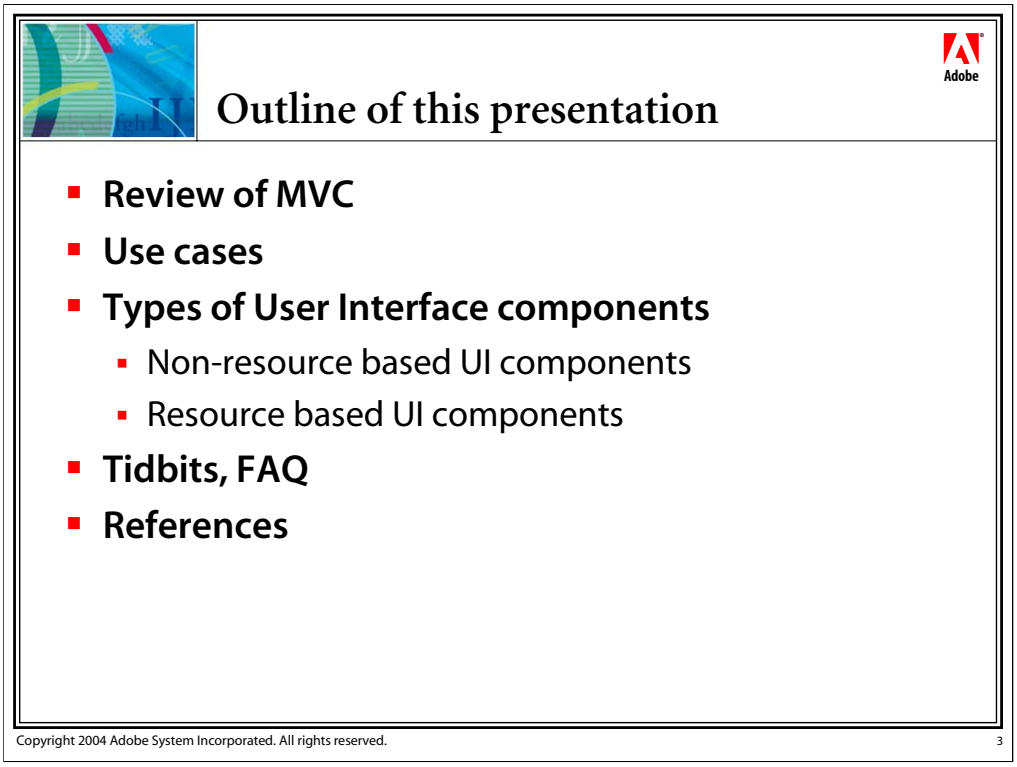
The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Adobe, Adobe InCopy and Adobe InDesign are trademarks of Adobe Systems Incorporated that may be registered in certain jurisdictions. Macintosh and Apple are registered trademarks, and Mac OS is a trademark of Apple Computer, Inc. Microsoft, Windows, Windows 95, Windows 98, Windows NT and Windows XP are registered trademarks of Microsoft Corporation. All other products or name brands are trademarks of their respective holders.



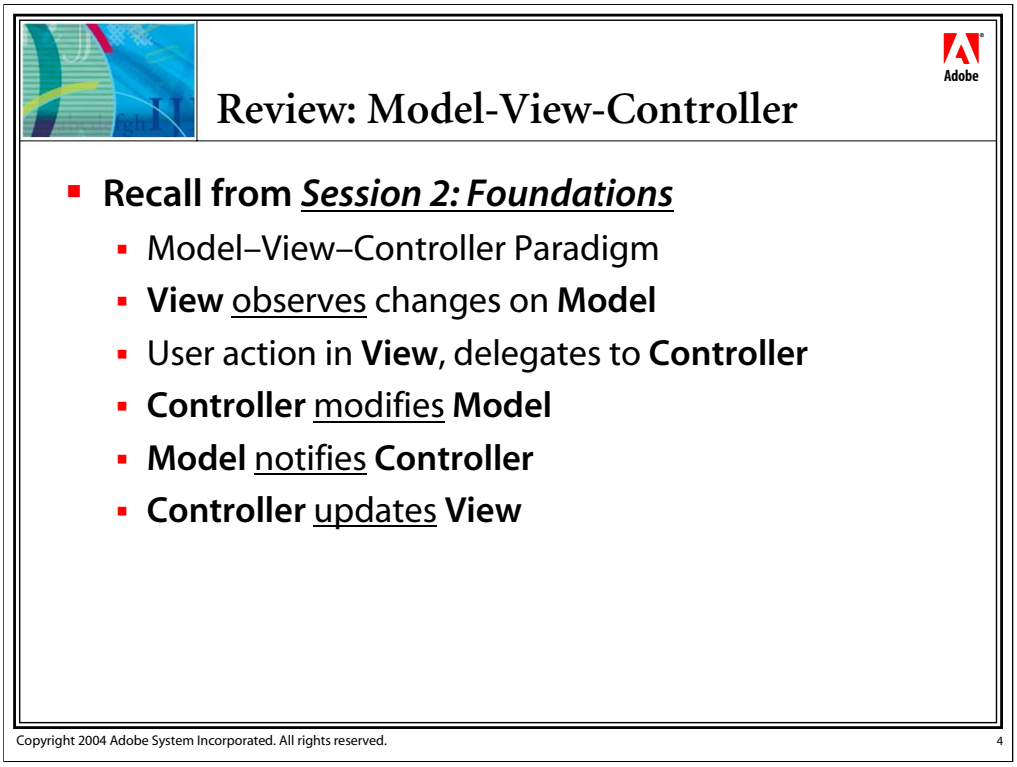
Today's session is titled "User Interface", and is the last session in our series. In this session, we will give an overview of how to create a user interface for your plug-ins.

We have covered quite a few fundamental topics that are necessary to understand when developing plug-ins for InDesign and InCopy. When you implement a user interface on a plug-in, you will need to know the basics of the object model and messaging, and a little bit about selection, which we have already covered.

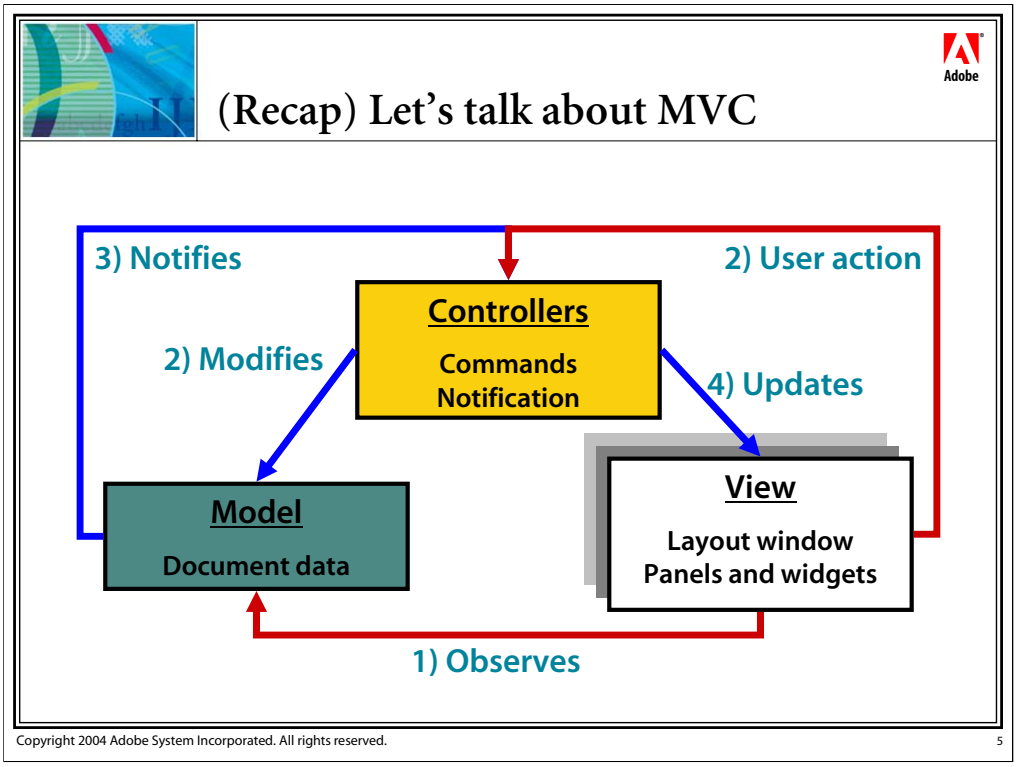


Specifically in this session, we will again review our MVC diagram, and see where the user interface fits in. Then we will present the use cases for user interfaces (UI), and describe the types of UI components you can reuse and/or implement. Along the way, we will discuss implementation roadmaps.

At the end, we will review the references for further exploration of the user interface subsystem.



Let's recall a slide from Session 2: Foundations, where we described the interaction between Model, View and Controller.

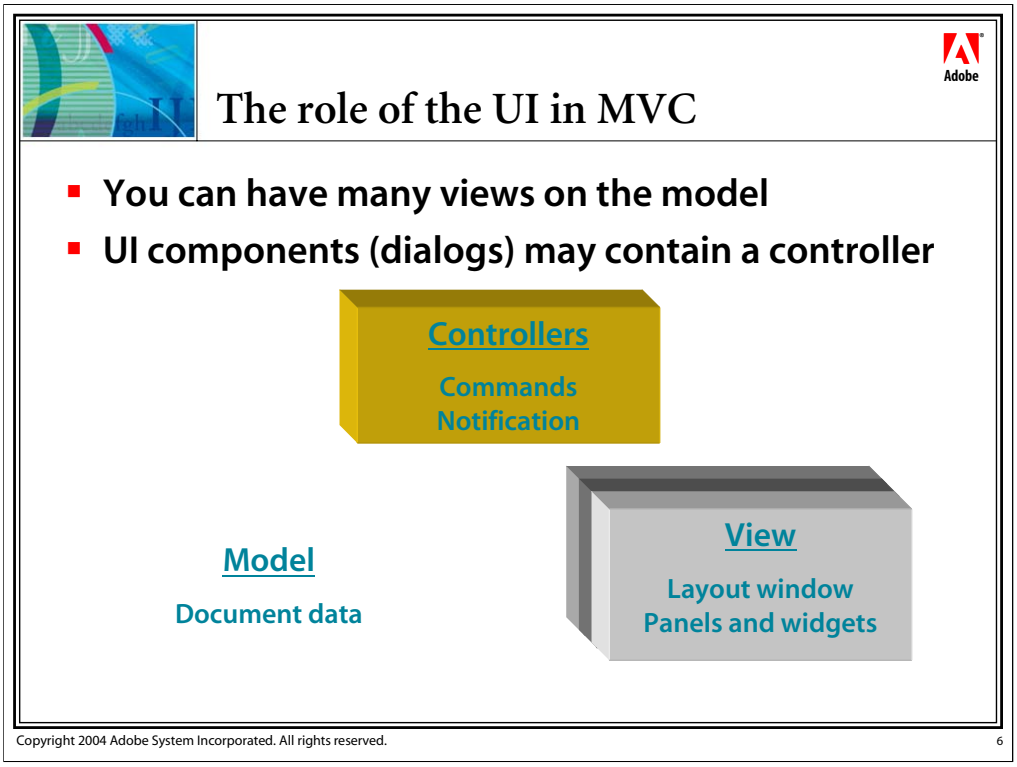


This slide is taken from Session 2: Foundation. Refer to Session 2 for details on this slide.

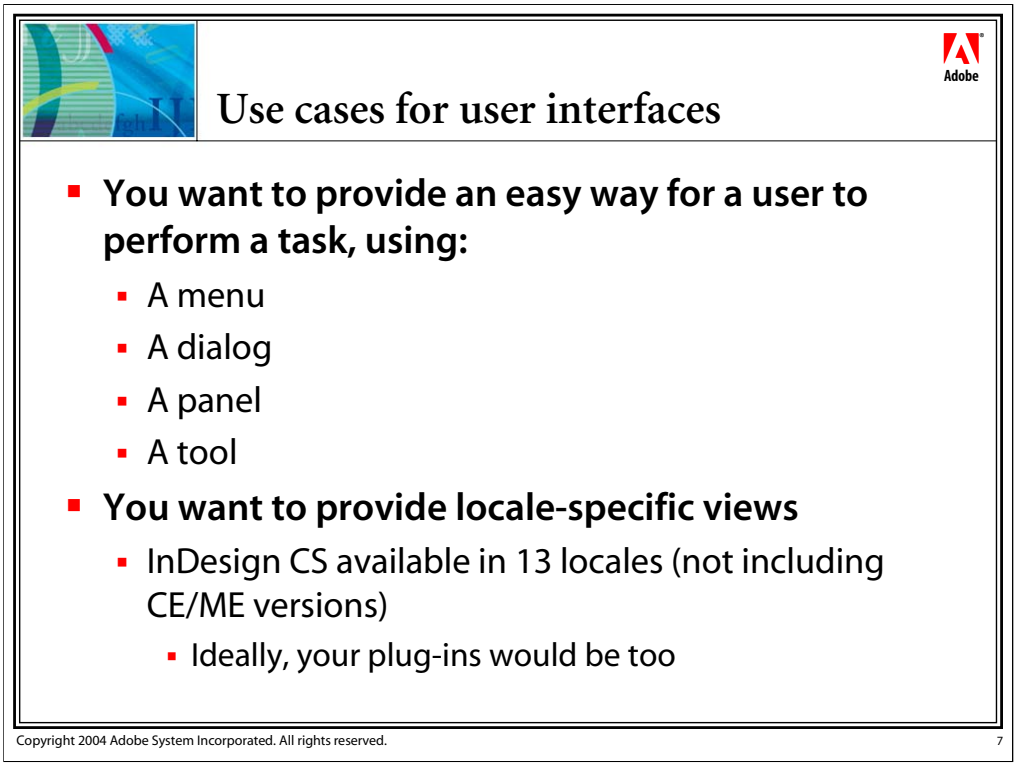
The red lines indicate the responsibilities of the view components, namely:

(1) Observes: The view components attach observers so they could be notified of changes, whether they be from the model or other widgets. As a result of this, the view components are updated when a change (of interest) occurs.

(2) User action: The actions of the users, such as mouse clicks, are captured in ways specific to the view, and then translated into some action that the controller could perform.



The role of the user interface components in InDesign/InCopy is to provide the view components in the MVC paradigm, and in some cases (such as dialog implementations), the controller implementations. Because of this paradigm, it is very easy to add multiple views to the same model data. An example was given in Session 2: Foundation, where we looked at how the Character panel and the Character/Paragraph toolbar both observe any changes to text attributes at currently selected text. If you change an attribute, say the point size of the selected text, using the Character panel, (1) the text layout reflects the change, and (2) the Character/Paragraph toolbar does as well. In this example, the layout window, the Character panel, and Character/Paragraph toolbars are all views that represent the text attributes in different ways.


Use cases for user interfaces
You want to provide an easy way for a user to perform a task, using:
A menu
A dialog
A panel
A tool
You want to provide locale-specific views
InDesign CS available in 13 locales (not including CE/ME versions)
Ideally, your plug-ins would be too
Copyright 2004 Adobe System Incorporated. All rights reserved.
7

The primary reason that we want to create a user interface is to provide an "easy" way for a user to perform a task. Discussing "Ease of use" could lead to an entirely different discussion, however, what we are interested in as a plug-in developer is to use the same UI factorizations that are available in other parts of the applications, namely:

•Menus

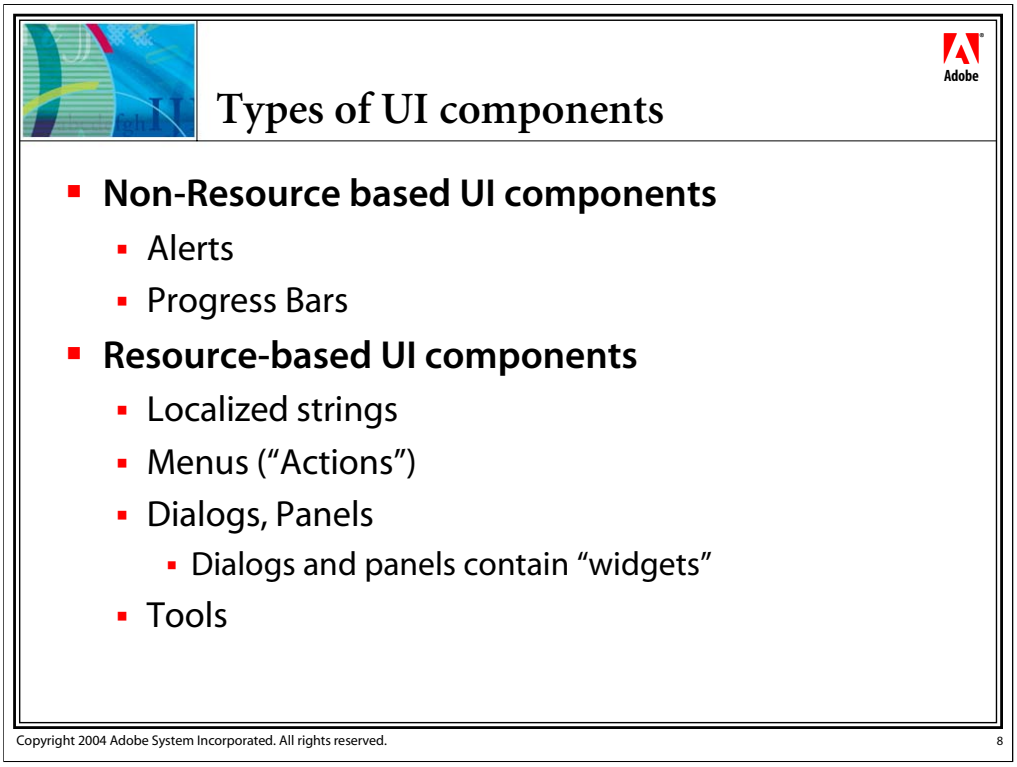
•Dialogs

•Panels (stash palettes and tabbed palettes are windows that contain panels. The dock bar, the tool box and kits are types of a panels)

•Tools

A secondary reason is to provide these user interfaces in locale-specific presentations. Adobe provides the InDesign CS application in 13 locales (not including the Central European and Middle Eastern versions). By providing support for these locales in your plug-ins' user interfaces, not only will you be eliminating language barriers for users around the world, you can provide locale-specific widget settings (to allow for different string lengths).

An application setting similar to the locale is the feature set, which allows you to provide different user experiences between users of the Japanese version and Roman locale versions (other languages). (See {SDK}/source/public/includes/FeatureSets.h for different feature set values).

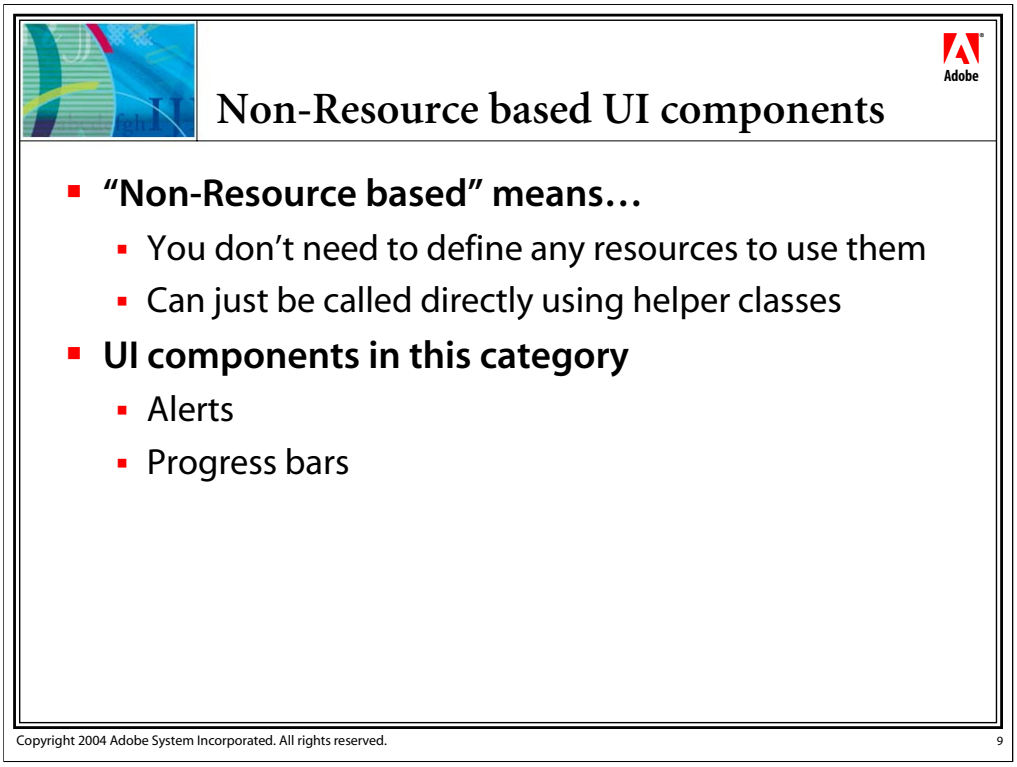


Here are some categories of UI components that you can use to satisfy the use cases from the previous slide. We will categorize them by whether or not you need to define resources to use them.

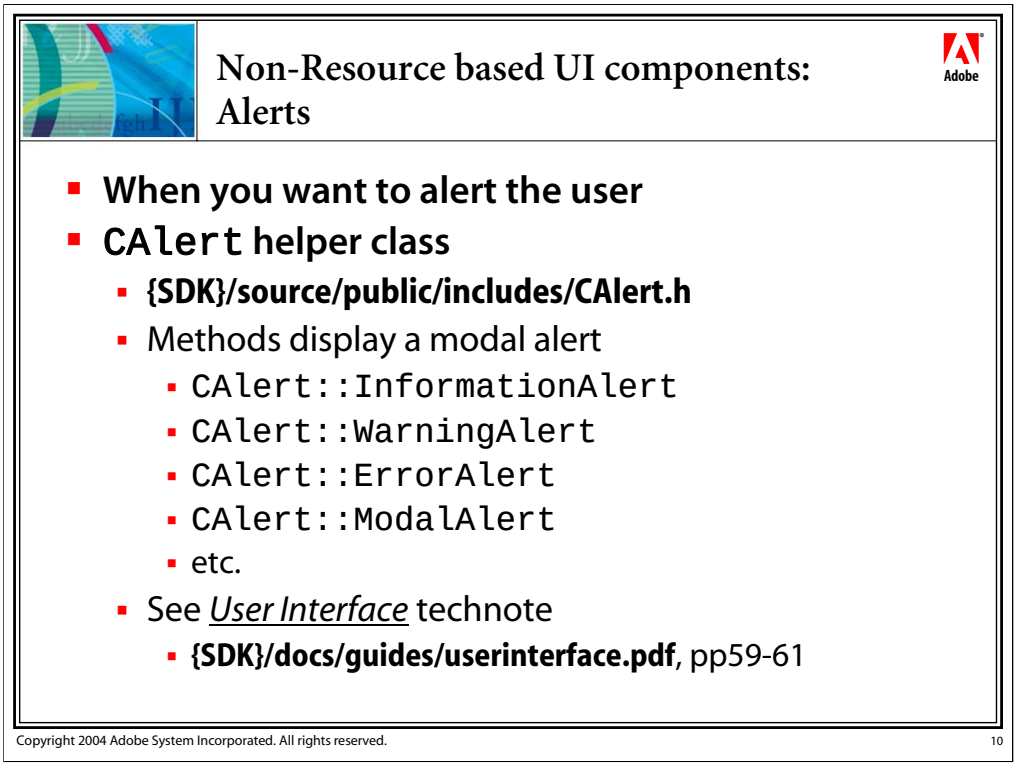
•Non-Resource based UI components are components you can use just by calling APIs in your C++ code.

•Resource-based UI components are components which require you to specify resources as well as implementing C++ code according to a specified framework.

The UI components which fall into each category are on the slide.


Non-Resource based UI components
"Non-Resource based" means…
You don't need to define any resources to use them
Can just be called directly using helper classes
UI components in this category
Alerts
Progress bars
Copyright 2004 Adobe System Incorporated. All rights reserved.
9

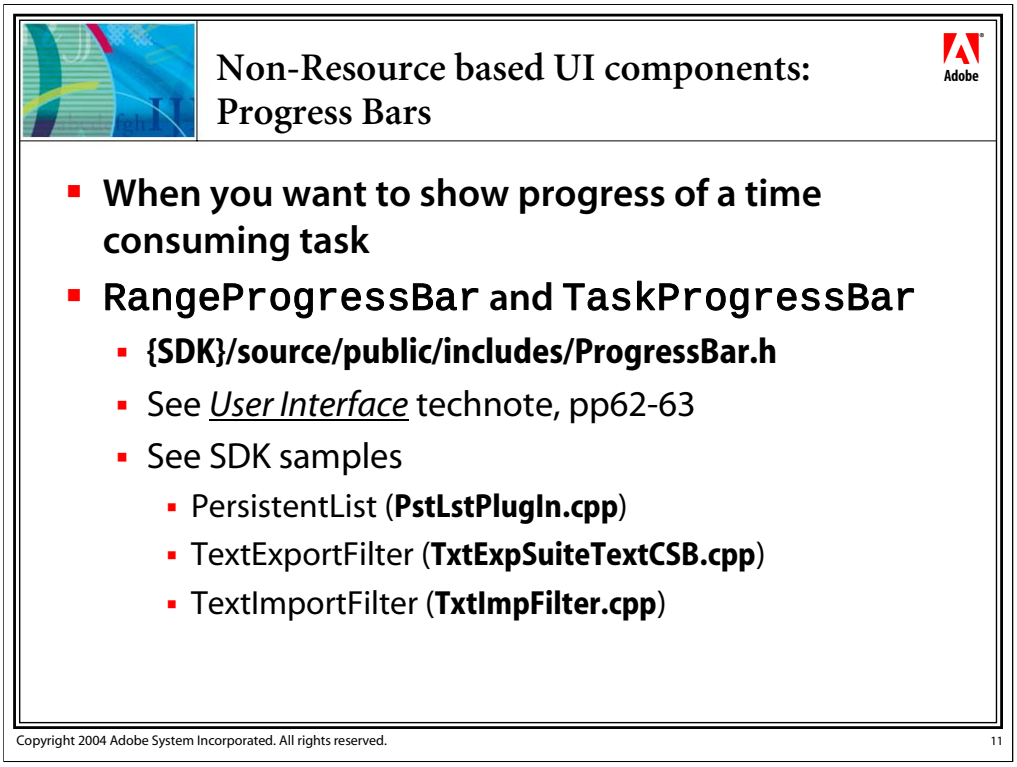
The non-resource based UI components include alerts and progress bars. The next few slides will explain them in more detail.



Alerts are used when you want to alert the user with a message, and a choice of how to proceed (usually "OK" and "Cancel"). Alerts can be displayed by calling static methods on the CAlert helper class. The slide shows some of the methods you can call.

There are many SDK samples that use CAlert.

Alerts can be customized in various ways, for instance, the "Do not show again" check box state can be stored so that you don't show the same alert, or an implementation of IAlertHandler (a service provider, kAlertHandlerService) can be used to override alert boxes. However, these are advanced topics that we won't cover in this session. Please refer to the CAlert.h and IAlertHandler.h header files for more details.



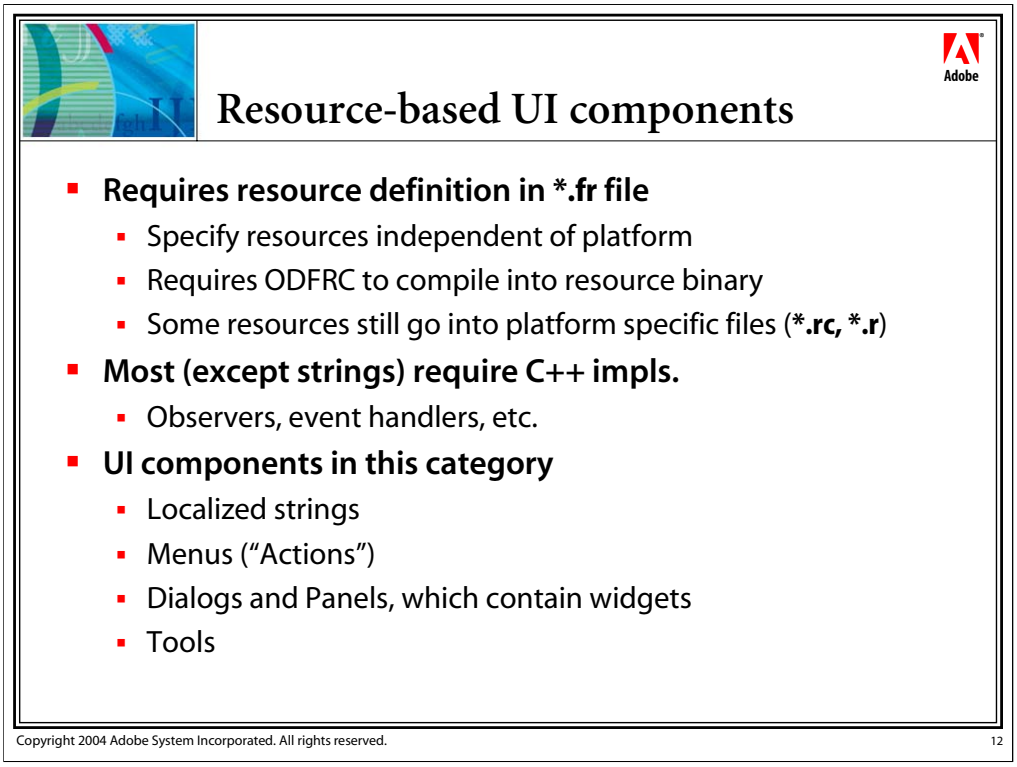
Progress bars are used when you want to show the progress of a task that takes a long time. By having a progress bar, the user isn't left wondering if the application has entered an endless loop.

There are two types of progress bars (both are subclasses of BaseProgressBar) you can use:

•RangeProgressBar: Shows progress based on a numeric scale (for example, "1 of 45")

•TaskProgressBar: Shows names of tasks being performed

The *User Interface* technote includes some discussion of ProgressBars, however, it goes into detail of the resources and implementations behind the two BaseProgressBar subclasses mentioned above. You can also refer to the SDK samples shown on the slide that use ProgressBars.


Resource-based UI components
Requires resource definition in *.fr file
Specify resources independent of platform
Requires ODFRC to compile into resource binary
Some resources still go into platform specific files (*.rc, *.r)
Most (except strings) require C++ impls.
Observers, event handlers, etc.
UI components in this category
Localized strings
Menus ("Actions")
Dialogs and Panels, which contain widgets
Tools
Copyright 2004 Adobe System Incorporated. All rights reserved.
12

The resource-based UI components require resource definitions in your plug-in project's *.fr file.

If you are implementing a regular Windows or Macintosh graphical user interface application, you would specify various UI resources (dialogs, menus, strings, etc.) in a compiler-specific resource format in *.rc files (Win) or *.rsrc files (Mac). However, by utilizing ODFRC, resources for InDesign/InCopy plug-ins can be specified in a platform-independent syntax, which means you only need to write the resource definitions once. The application's user interface subsystem provides native support for common UI elements. One example is the use of buttons from the Aqua user interface subsystem of Mac OS X. Platform-specific operating system events, such as key presses and mouse clicks, are also abstracted so that your plug-in code can handle events in a common fashion.

NOTE: Some resources, such as icons, still go into platform specific resource files (*.rc, *.r). However, these can be referenced by resource IDs from UI resources in *.fr files.

Most UI components (with the exception of localizable strings) require you to write C++ implementations according to the user interface framework, including observers, event handlers, etc. These implementations would be placed in boss classes that correspond to the resource definitions of the UI component.

The UI components that fall into this category are:

•Localized strings

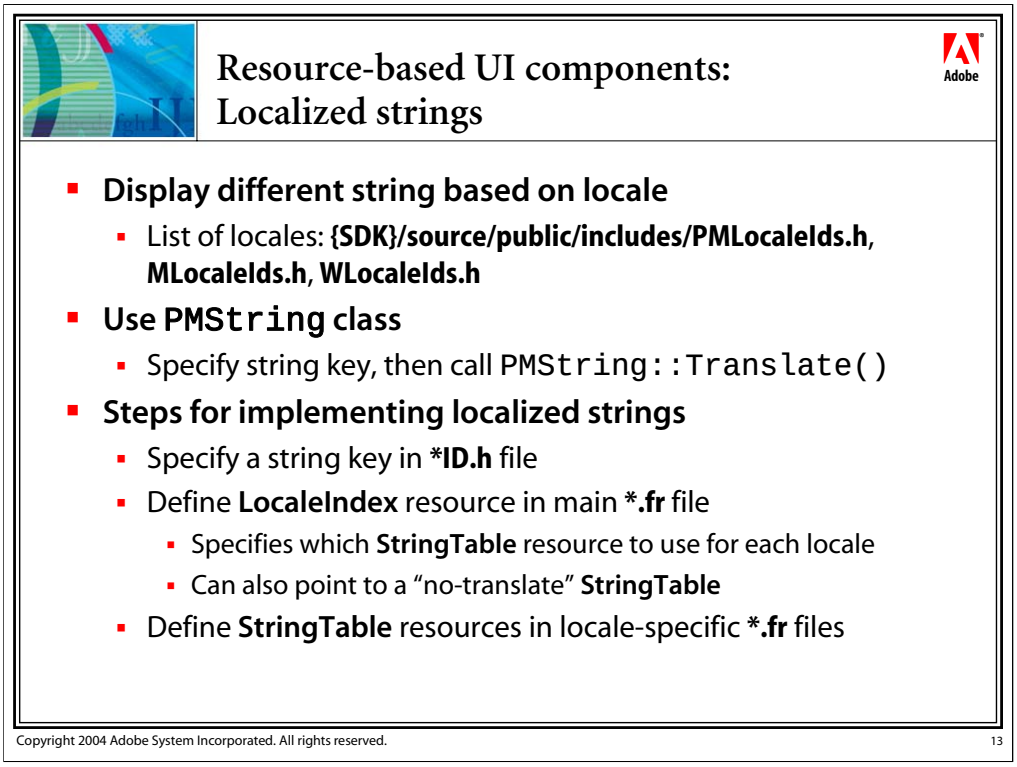
•Menus (or Actions)

•Dialogs

•Panels

•Tools

The next few slides will explain them in more detail.


Resource-based UI components:
Localized strings
Adobe
Display different string based on locale
List of locales: {SDK}/source/public/includes/PMLocaleIds.h, MLocaleIds.h, WLocaleIds.h
Use PMString class
Specify string key, then call PMString::Translate()
Steps for implementing localized strings
Specify a string key in *ID.h file
Define LocaleIndex resource in main *.fr file
Specifies which StringTable resource to use for each locale
Can also point to a "no-translate" StringTable
Define StringTable resources in locale-specific *.fr files
Copyright 2004 Adobe System Incorporated. All rights reserved.
13

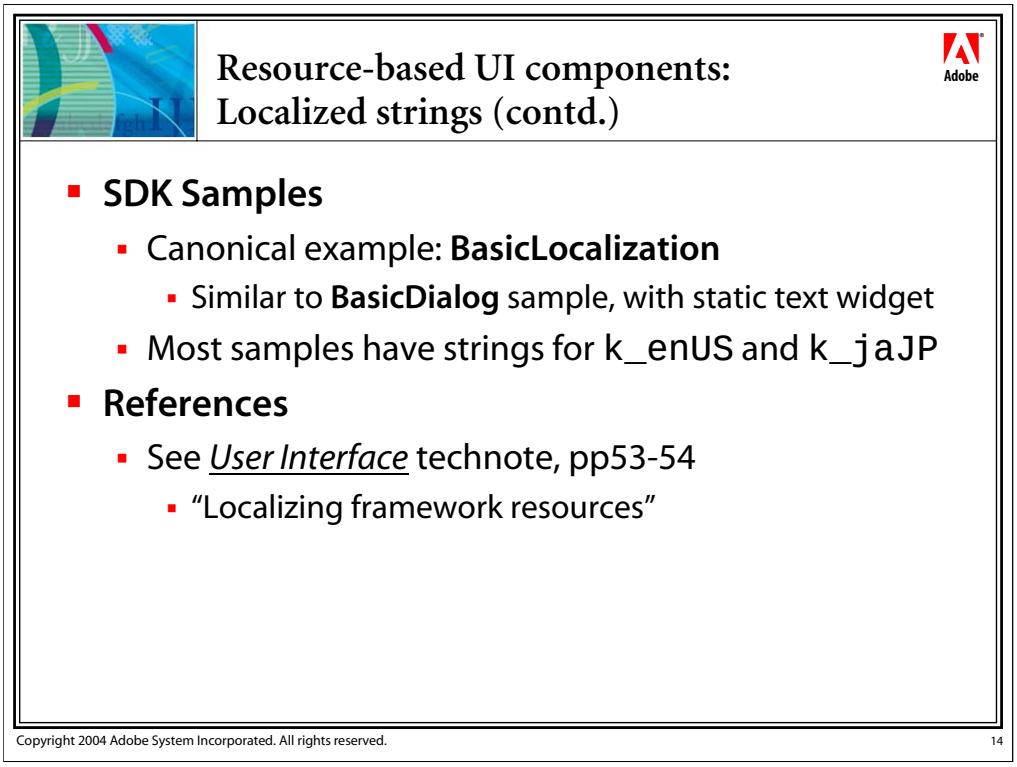
Localized strings are used to display strings on the user interface based on a locale. Locales are identified by codes such as k_enUS (US English), k_jaJP (Japanese), etc. and represent the user interface language.

Localized strings are represented by the PMString class. PMString utilizes the application's string table to obtain a locale-specific string based on a "string key". Conversely stated, if you want to get some string in a specific locale, you create an instance of PMString by specifying a string key in the constructor, then call the PMString::Translate() method. You can then obtain a CString representation of the string by calling PMString::GrabCString(), or a Unicode (UTF-16) representation of the string by calling PMString::GrabUTF16Buffer().

Any string that is displayed on a user interface component is a PMString. Even strings on alerts and progress bars use PMString.

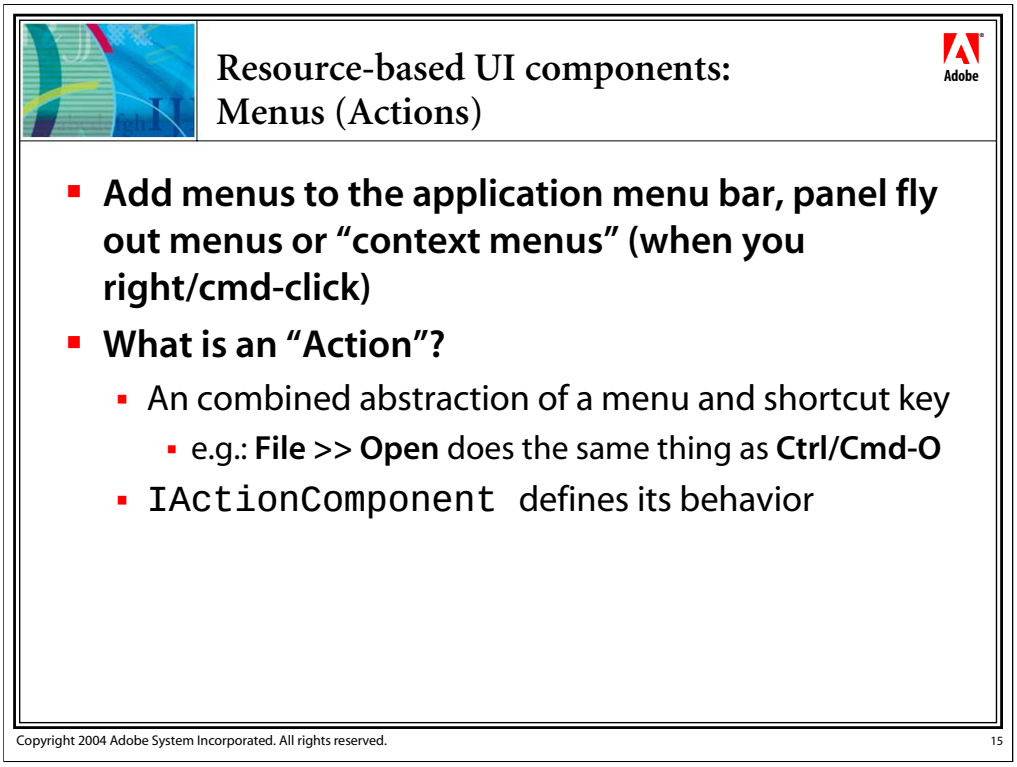
Implementation steps are as follows:

1. Specify a string key in your plug-in project's ID.h file. The string key is used in to identify a unique string in the application's string table (hash table). The hash function is influenced by the first and last letters of the string key. String key collisions are rare, however, we have seen a few cases in SDK samples in previous versions, since most SDK sample string keys start with the "0" (zero, first number in a hex representation of the plug-in prefix ID) and end in "y" (last letter in "Key").
2. Define a LocaleIndex in your plug-in's main *.fr file. This LocaleIndex resource tells the application which string table to refer to for specific locales and feature sets. You can also have a LocaleIndex resource that specifies to look at a single StringTable for all locales and feature sets (we call it a "no-translate" string table).
3. Define StringTable resources in each of your locale-specific *.fr files (e.g. WFP_enUS.fr, WFP_jaJP.fr). The StringTable resource provides mappings between string keys and their locale-specific string translations.



Any SDK sample with any kind of user interface will have a set of localized strings, as we provide US English and Japanese strings for all plug-ins. There is one sample that provides strings for all 13 locales and feature sets (except for the combination of the Japanese feature set and non-Japanese languages), and that is BasicLocalization. The plug-in code is very similar to the BasicDialog sample plug-in (to be discussed in a later slide), however, there is a static text widget on the dialog. Strings on dialogs and panels get automatically translated, as dialogs and panels are created based on a specific locale, so the UI subsystem also knows which locale use for translating the strings.

Further details about localized strings can be found in the *User Interface* technote, as shown on the slide.


Resource-based UI components:
Menus (Actions)
Adobe
Add menus to the application menu bar, panel fly out menus or "context menus" (when you right/cmd-click)
What is an "Action"?
An combined abstraction of a menu and shortcut key
e.g.: File >> Open does the same thing as Ctrl/Cmd-O
IActionComponent defines its behavior
Copyright 2004 Adobe System Incorporated. All rights reserved.
15

Menus allow for categorization of user actions, and can be located at the top of the application window, the flyout menu button on panels, or context menus (when you right-click or cmd-click somewhere in the application)

Users of InDesign/InCopy applications can invoke menus by pressing shortcut keys (e.g. CTRL-Q to quit), which is the basis for abstracting menus and shortcut keys as "actions". The implementation of IActionComponent defines the behavior behind a menu selection and shortcut key press.

The steps for implementing menus are shown on the slides.

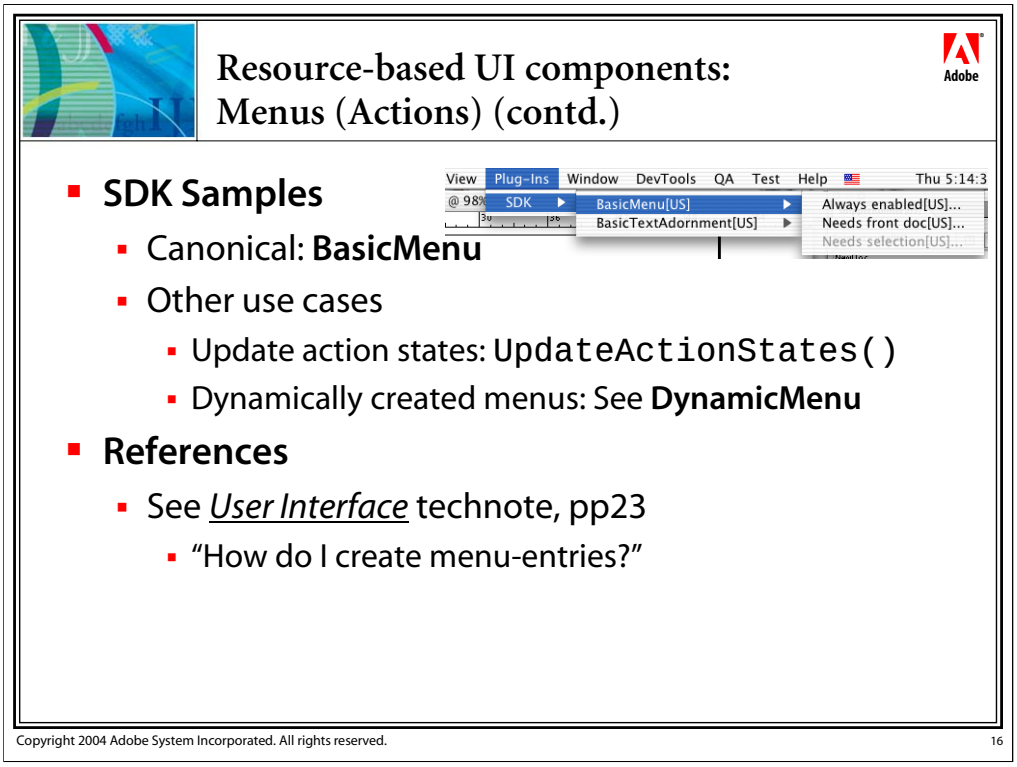
The resources that you must define are:

•ActionDef: specifies action parameters, such as the ActionID and the action component boss

•MenuDef: specifies menu parameters, such as the menu path, menu location index, etc.

•Menu strings: such as menu names. See details on localized strings

The C++ implementation you must write is a class that derives from IActionComponent (or the partial implementation CActionComponent). This implementation must be aggregated in a boss with an implementation of IPMPersist (you can reuse kPMPersistImpl). The IPMPersist interface is necessary since action parameters, along with other UI states, are persisted in the session database (We'll bring this up in the FAQ section towards the end of this presentation).



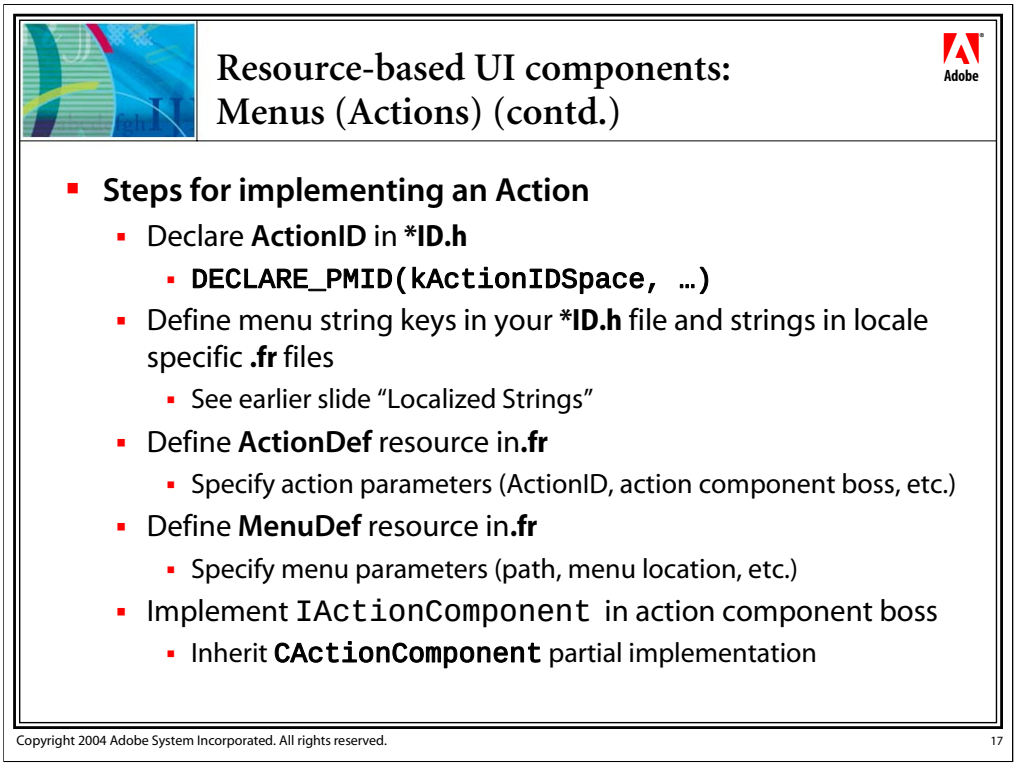
Many SDK samples have a menu (most have at least the About… menu). The canonical example is the BasicMenu plug-in, which shows how to create a menu, have the action state update itself automatically based on the current selection, and uses/implements a simple selection suite.
There are other use cases related to menus/actions.

•To update action states (checked, disabled, etc.) programmatically, you can implement the UpdateActionStates() method that is inherited from CActionComponent.

•To create menus dynamically, refer to the DynamicMenu sample plug-in.

If you want to implement menus from scratch, the following slide includes detailed steps on how to do just that.

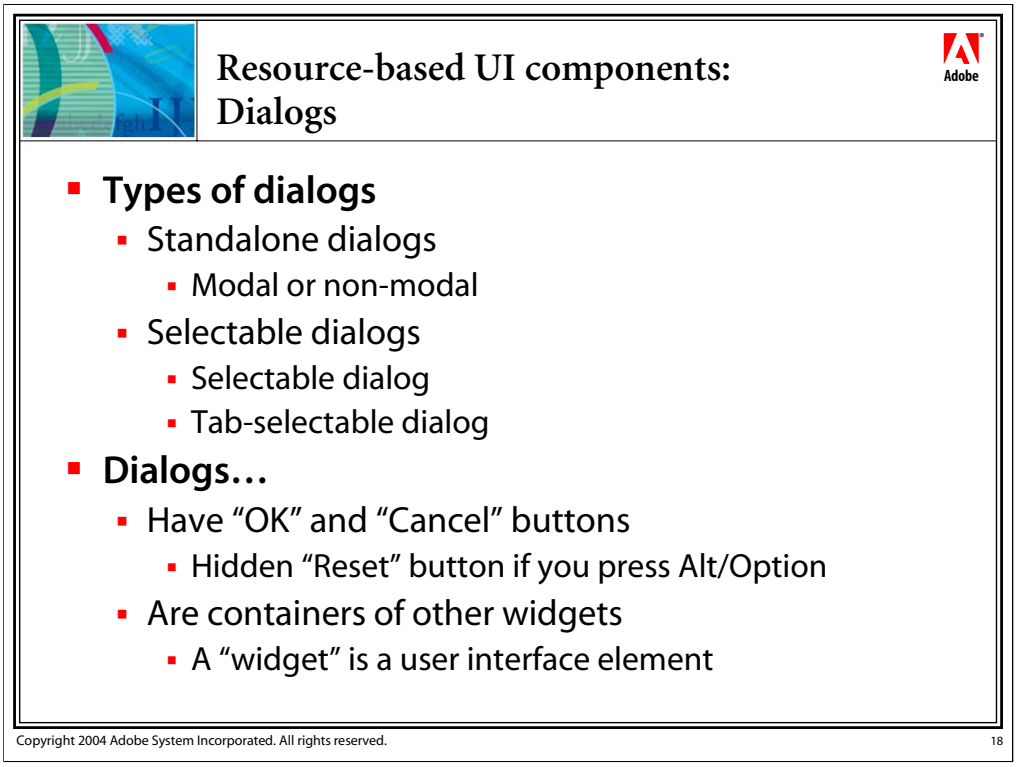
For more information on menus in general, refer to the *User Interface* technote, as shown on the slide.


Resource-based UI components:
Menus (Actions) (contd.)
Adobe
Steps for implementing an Action
Declare ActionID in *ID.h
DECLARE_PMID(kActionIDSpace, …)
Define menu string keys in your *ID.h file and strings in locale specific .fr files
See earlier slide "Localized Strings"
Define ActionDef resource in.fr
Specify action parameters (ActionID, action component boss, etc.)
Define MenuDef resource in.fr
Specify menu parameters (path, menu location, etc.)
Implement IActionComponent in action component boss
Inherit CActionComponent partial implementation
Copyright 2004 Adobe System Incorporated. All rights reserved.
17

Here are the steps for implementing actions.

1. Declare an ActionID in your ID.h file. Use the DECLARE_PMID macro with kActionIDSpace.
2. Define menu strings in your ID.h file. These strings are used as menu display strings and action areas (for grouping shortcut keys), and are localized.
3. Define a single ActionDef resource in your .fr file for all actions/menus. This is where you specify action parameters, such as the ActionID, the action component boss class ID, etc. See {SDK} /source/public/includes/ActionDef.fh for the parameters of an ActionDef.
4. Define a single MenuDef resource in your .fr file for all menus. This is where you specify menu parameters, such as the menu path, menu location within a sub menu, etc. The menu path starts with "Main", "Rt" (for context menus – see below), or the string specified in a panel widget resource. Use a colon (":") to delimit menu paths, and ":-" to add a separator.
    - Context menus are no different than regular menus – you just need to know the menu path. The menu paths for context menus usually begin with "Rt***":
        - "RtMouseText" for right/cmd-click on text
        - "RtMouseTable" for right/cmd-click in a table
        - "RtMouseLayout" for right/cmd-click on a layout
        - "RtMouseGuide" for right/cmd-click on a guide
        - "RtMouseNote" for right/cmd-click on an InCopy note
        - "RtMouseDefault" for right/cmd-click when there is no document open

Implement IActionComponent (you can inherit the partial implementation CActionComponent), and aggregate this in an action component boss. The method you must implement at the least is DoAction(IActiveContext* ac, ActionID actionID, GSysPoint mousePoint = kInvalidMousePoint, IPMUnknown* widget = nil). In the same boss, also aggregate IID_IPMPERSIST kPMPersistImpl, since action states are persisted.



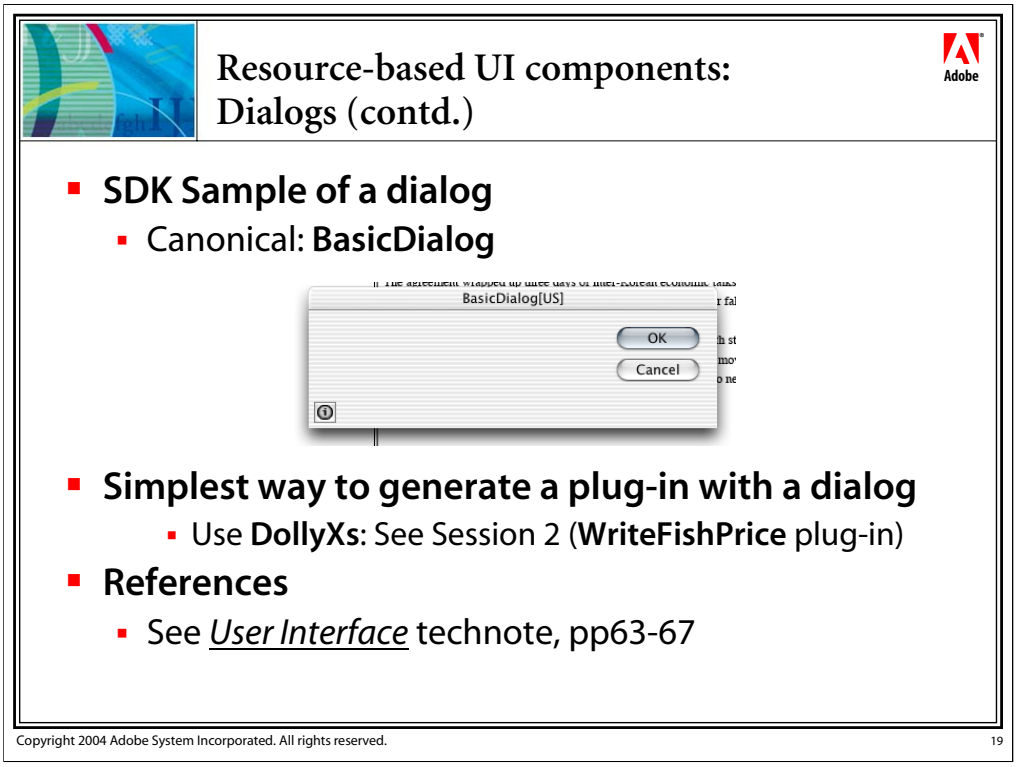
The dialog is another type of resource-based UI component available in the user interface subsystem. There are two types of dialogs:

•Standalone dialogs, which can be opened in a modal or non-modal way.

•Selectable dialogs, which are housed in a parent dialog and can be individually selected from a set of tabs or entries in a list.

Dialogs all have at least two buttons: "OK" and "Cancel". Also, if you press the Alt-key (Windows) or Option-key (Mac), the "Cancel" button turns into a "Reset" button, which allows you to reset the settings on the dialog to their default values.

Dialogs in general are containers of widgets, which are user interface elements. We will discuss widgets in greater detail later. A dialog is also a widget itself. (We'll see why this is useful)

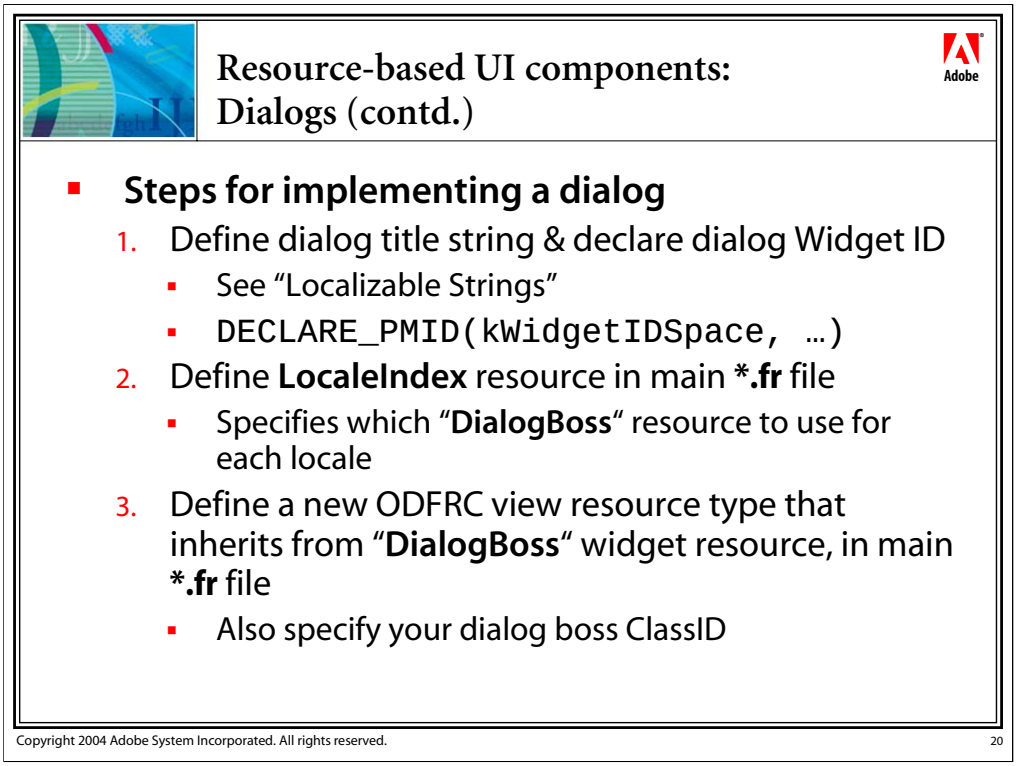


The canonical example is the BasicDialog plug-in, which shows how to implement a dialog, and opens the dialog from a menu (so it also has an IActionComponent implementation). If you look at the BscDlgActionComponent::DoDialog() private method, you can see how the dialog is opened, using IDialogMgr. The IDialogMgr::CreateNewDialog() method call is where you specify the modality of the dialog.

The simplest way to implement a plug-in with a dialog is to use DollyXs to generate such a plug-in. If you recall from Session 2: Foundations, we reviewed the WriteFishPrice plug-in, which was generated with those very steps. Please refer to the *Porting Guide* technote ({SDK}/docs/guides/portingguide.pdf) for details on how to use **DollyXs**.

However, if you want to implement a dialog from scratch, the following slides include detailed steps of how to do just that.

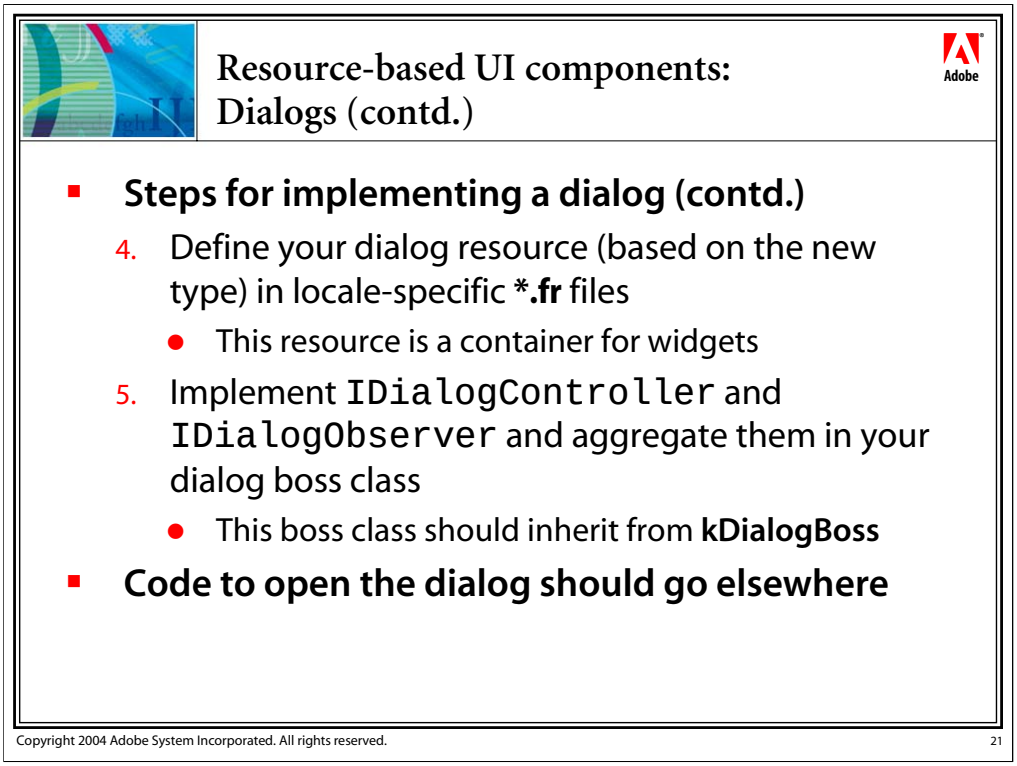
For further information on dialogs in general, refer to the *User Interface* technote.


Resource-based UI components:
Dialogs (contd.)
Adobe
Steps for implementing a dialog
1. Define dialog title string & declare dialog Widget ID
▪ See "Localizable Strings"
▪ DECLARE_PMID(kWidgetIDSpace, …)
2. Define LocaleIndex resource in main *.fr file
▪ Specifies which "DialogBoss" resource to use for each locale
3. Define a new ODFRC view resource type that inherits from "DialogBoss" widget resource, in main *.fr file
▪ Also specify your dialog boss ClassID
Copyright 2004 Adobe System Incorporated. All rights reserved.
20

[Presenter may skip this slide] Here are the steps in implementing a dialog (modal or non-modal).

1. Define a dialog title string and declare a dialog WidgetID. The string is created in the same way as defining a localizable string, since dialogs are generally specified in a locale-specific way. The WidgetID is declared in the kWidgetIDSpace using DECLARE_PMID.
2. Define a LocaleIndex resource in your plug-in's main *.fr file. This resource is a mapping between locales and different "DialogBoss" resources, which describe the visual specification of a dialog. (NOTE: The name "DialogBoss" doesn't refer to a boss class, but the widget name. This name is kept for historical reasons.) Using this LocaleIndex resource, you can have slightly different visual specifications of dialogs per locale. For instance, German strings tend to be 50% longer than English strings on average, so text widgets may require extra space on a dialog. In this case, you might decide to have a different visual specification on the DialogBoss resource in the k_deDE locale.
   NOTE: If you look at the LocaleIndex (kSDKDefDialogResourceID) resource in SDK sample plug-ins with dialogs, you will notice that they do not use locale-specific DialogBoss resources. Instead, they use the same DialogBoss resource within the plug-in.
3. Define a new ODFRC view resource type that inherits from the DialogBoss widget resource. You would specify this once in your plug-in's main *.fr file. This is also where you also specify the actual boss class that provides the implementation behind the DialogBoss widget resource. This statement looks sometihng like this (sample from BasicDialog):
   type BscDlgDialogBoss(kViewRsrcType) : DialogBoss(ClassID = kBscDlgDialogBoss) {};
   kBscDlgDialogBoss is the boss class that provides the implementations for the BscDlgDialogBoss dialog widget resource.

(continued on next slide)


Resource-based UI components:
Dialogs (contd.)
Adobe
Steps for implementing a dialog (contd.)
4. Define your dialog resource (based on the new type) in locale-specific *.fr files
This resource is a container for widgets
5. Implement IDialogController and IDialogObserver and aggregate them in your dialog boss class
This boss class should inherit from kDialogBoss
Code to open the dialog should go elsewhere
Copyright 2004 Adobe System Incorporated. All rights reserved.
21

[Presenter may skip this slide]

4. Define your dialog resource based on the new DialogBoss type that you just defined. You would do this in your locale-specific *.fr file. The DialogBoss resource is a container for other widgets (not the DialogBoss resource itself), so within this resource, you would specify other child resources. We will discuss these widgets in a later slide.

5. Implement IDialogController (using the partial implementation CDialogController) and IDialogObserver (using the partial implementation CDialogObserver), and aggregate them in your dialog boss class, when you declare it in the ClassDescriptionTable resource. This boss class should inherit from kDialogBoss, since your widget resource inherits from DialogBoss. Here's what the boss class declaration would look like (sample from BasicDialog):

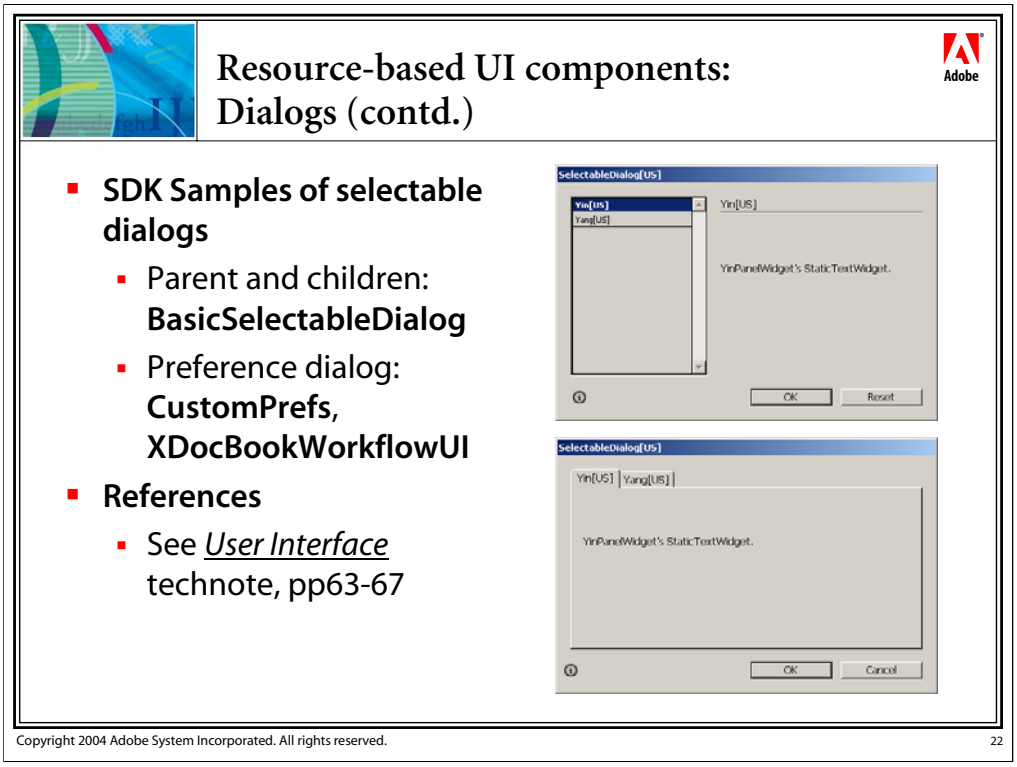
```
Class {
 kBscDlgDialogBoss,
 kDialogBoss, {
  IID_IDIALOGCONTROLLER, kBscDlgDialogControllerImpl,
  IID_IOBSERVER, kBscDlgDialogObserverImpl, } },
```

In your IDialogController implementation, you would add the code to initialize the dialog widgets, validate the dialog widgets when the "OK" button is pressed, and apply the data from the dialog widgets after the validation passes (after "OK" is pressed). In your IDialogObserver implementation, you would add the code to attach the observer for child widgets (in the AutoAttach() method), detach the observer (in the AutoDetach() method), and the actual responses (in the Update() method).

•NOTE: We had a developer case where a developer wondered why he could use an accelerator key (a button with an underlined character that is identified by an "&" in the string) for a button widget on Windows, but not on Mac. Using accelerator keys for button widgets is a Windows-only feature, and is generally discouraged for widgets.

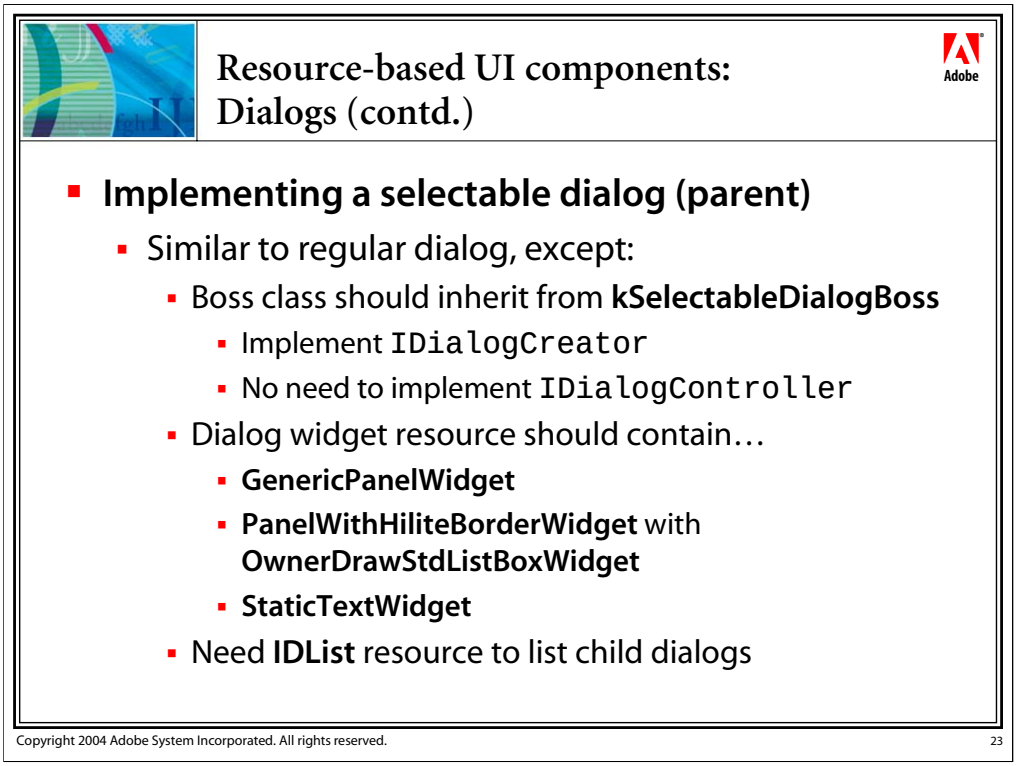
That concludes the process of implementing a dialog.

The code to open this dialog should go elsewhere.



The canonical example of a selectable dialog is the BasicSelectableDialog plug-in, which shows how to implement two different types of selectable dialogs and provides the child dialogs as well. There are other samples that provide just the child dialog (for the Preferences dialog): CustomPrefs and XDocBookWorkflowUI.

For further references, refer to the Dialogs section in the *User Interface* technote.

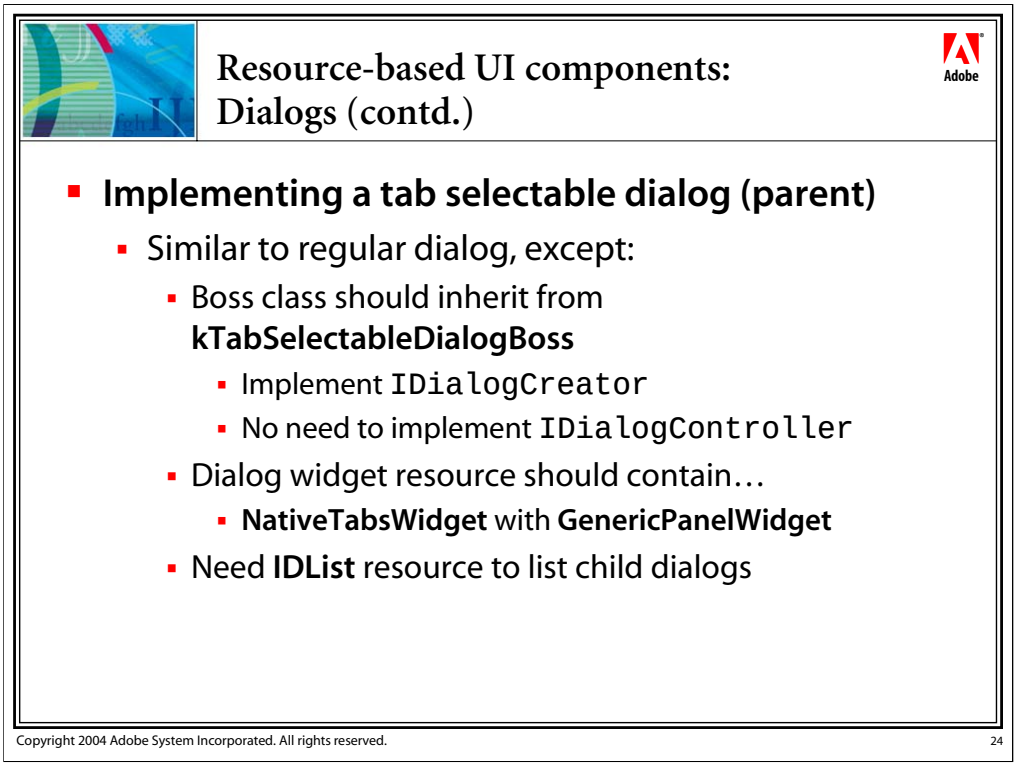


[Presenter may skip this slide] The steps to implement a selectable dialog parent (which is the dialog that houses the child dialogs that are selectable from a list) is similar to a regular dialog, except:

•The boss class should inherit from kSelectableDialogBoss
You need to provide your own implementation of IDialogCreator by inheriting the partial implementation CDialogCreator, so that the child dialogs can be created when they are selected. The methods you need to implement are:
* CreateDialog(): This is where you use IDialogMgr::CreateNewDialog
* GetOrderedPanelIDs(): Enables resource access and delegates to CDialogCreator::GetOrderedPanelIDs()
* GetOrderedPanelsRsrcID(): called by CDialogCreator::GetOrderedPanelIDs() to get the resource ID of the IDList resource (see below)
You don't need to implement IDialogController, since kSelectableDialogBoss already has one, unless you want to provide customized Initialization/Validation/Apply behaviors.
Optionally, you can implement IDialogObserver if you add widgets on the parent dialog.

•The Dialog widget resource should contain:
* GenericPanelWidget: to house the child dialog
* PanelWithHiliteBorderWidget with OwnerDrawStdListBoxWidget inside: to display a list of child dialogs
* StaticTextWidget to show the name of the active child dialog

•And you need an IDList resource that lists the boss classes for the child dialogs. The resource ID can be defined as a #define statement in your *ID.h file. It is to be specified from your implementation of IDialogCreator.


Resource-based UI components:
Dialogs (contd.)
Adobe
Implementing a tab selectable dialog (parent)
Similar to regular dialog, except:
Boss class should inherit from kTabSelectableDialogBoss
Implement IDialogCreator
No need to implement IDialogController
Dialog widget resource should contain…
NativeTabsWidget with GenericPanelWidget
Need IDList resource to list child dialogs
Copyright 2004 Adobe System Incorporated. All rights reserved.
24

[Presenter may skip this slide] The steps to implement a tab selectable dialog parent (which is the dialog that houses the child dialogs that are selectable by clicking a tab) is similar to a regular dialog, except:

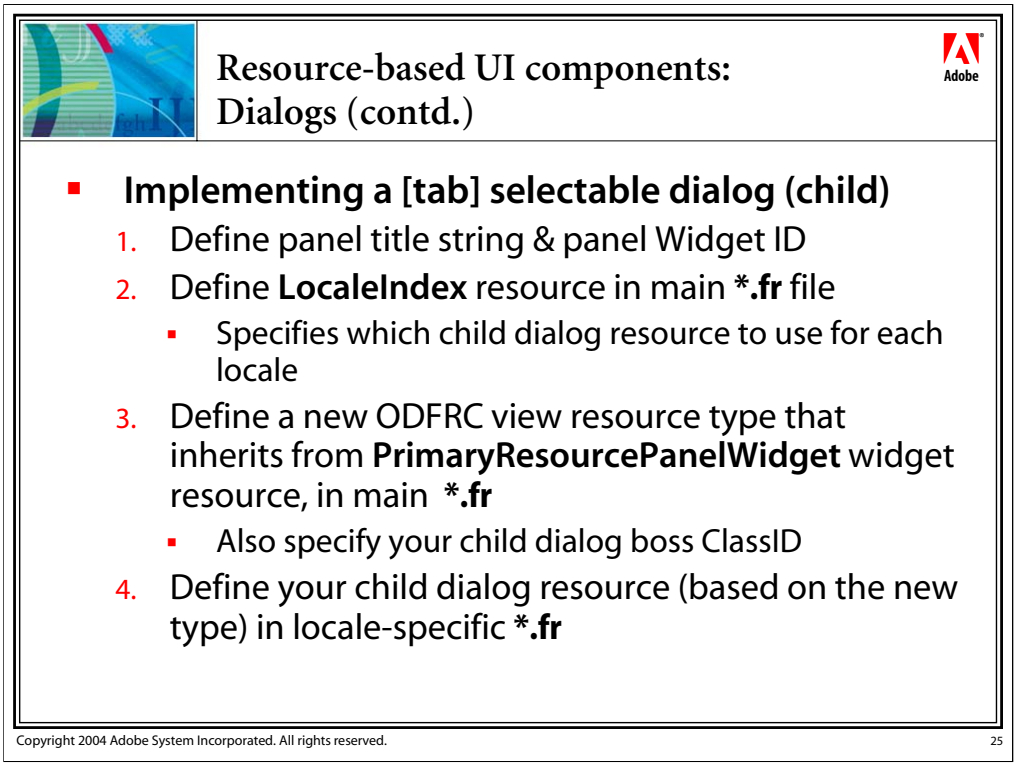
•The boss class should inherit from kTabSelectableDialogBoss

- •You need to provide your own implementation of IDialogCreator by inheriting the partial implementation CDialogCreator, so that the child dialogs can be created when they are selected. The methods you need to implement are:
  - •CreateDialog(): This is where you use IDialogMgr::CreateNewDialog
  - •GetOrderedPanelIDs(): Enables resource access and delegates to CDialogCreator::GetOrderedPanelIDs()
  - •GetOrderedPanelsRsrcID(): called by CDialogCreator::GetOrderedPanelIDs() to get the resource ID of the IDList resource (see below)
- •You don't need to implement IDialogController, since kTabSelectableDialogBoss already has one, unless you want to provide customized Initialization/Validation/Apply behaviors.
- •Optionally, you can implement IDialogObserver if you add widgets on the parent dialog.

•The Dialog widget resource should contain:

- •NativeTabsWidget with GenericPanelWidget: to display tabs and to house the child dialog

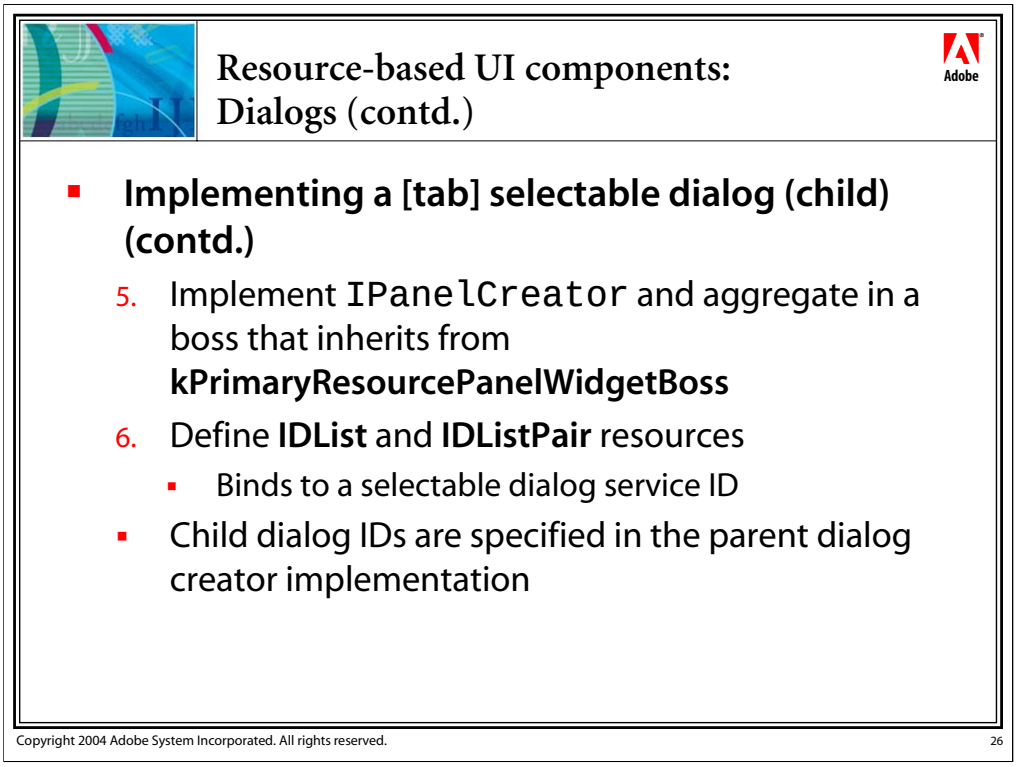
•And you need an IDList resource that lists the boss classes for the child dialogs. The resource ID can be defined as a #define statement in your *ID.h file. It is to be specified from your implementation of IDialogCreator.


Resource-based UI components:
Dialogs (contd.)
Adobe
Implementing a [tab] selectable dialog (child)
1. Define panel title string & panel Widget ID
2. Define LocaleIndex resource in main *.fr file
Specifies which child dialog resource to use for each locale
3. Define a new ODFRC view resource type that inherits from PrimaryResourcePanelWidget widget resource, in main *.fr
Also specify your child dialog boss ClassID
4. Define your child dialog resource (based on the new type) in locale-specific *.fr
Copyright 2004 Adobe System Incorporated. All rights reserved.
25

[Presenter may skip this slide] The steps to implement a selectable or tab selectable dialog child is as follows:

1. Define a panel title string and declare a panel widget ID. The string is created in the same way as defining a localizable string, since panels are generally specified in a locale-specific way. The WidgetID is declared in the kWidgetIDSpace using DECLARE_PMID.
2. Define a LocaleIndex resource. This resource is a mapping between locales and different panel resources, which describe the visual specification of a panel.
3. Define a new ODFRC view resource type that inherits from the PrimaryResourcePanelWidget widget resource. You would specify this once in your plug-in's main *.fr file. This is also where you also specify the actual boss class that provides the implementation behind the panel widget resource. This statement looks something like this (sample from BasicSelectableDialog):
   type YangPanelWidget(kViewRsrcType) : PrimaryResourcePanelWidget(ClassID = kYangPanelBoss) { };
   kYangPanelBoss is the boss class that provides the implementations for the YangPanelWidget resource.
4. Define your child dialog resource based on the new panel resource type that you just defined. You would do this in your locale-specific *.fr file. The panel resource is a container for other widgets so within this resource, you would specify other child resources. We will discuss these widgets in a later slide.

(Continued on next slide)


Resource-based UI components:
Dialogs (contd.)
Adobe
Implementing a [tab] selectable dialog (child) (contd.)
5. Implement IPanelCreator and aggregate in a boss that inherits from kPrimaryResourcePanelWidgetBoss
6. Define IDList and IDListPair resources
Binds to a selectable dialog service ID
Child dialog IDs are specified in the parent dialog creator implementation
Copyright 2004 Adobe System Incorporated. All rights reserved.
26

[Presenter may skip this slide]

5. Implement IPanelCreator (using the partial implementation CPanelCreator) and aggregate it in your child dialog boss class, when you declare it in the ClassDescriptionTable resource. This boss class should inherit from kPrimaryResourcePanelWidgetBoss, since your widget resource inherits from PrimaryResourcePanelWidget. Here's what the boss class declaration would look like (sample from BasicSelectableDialog):

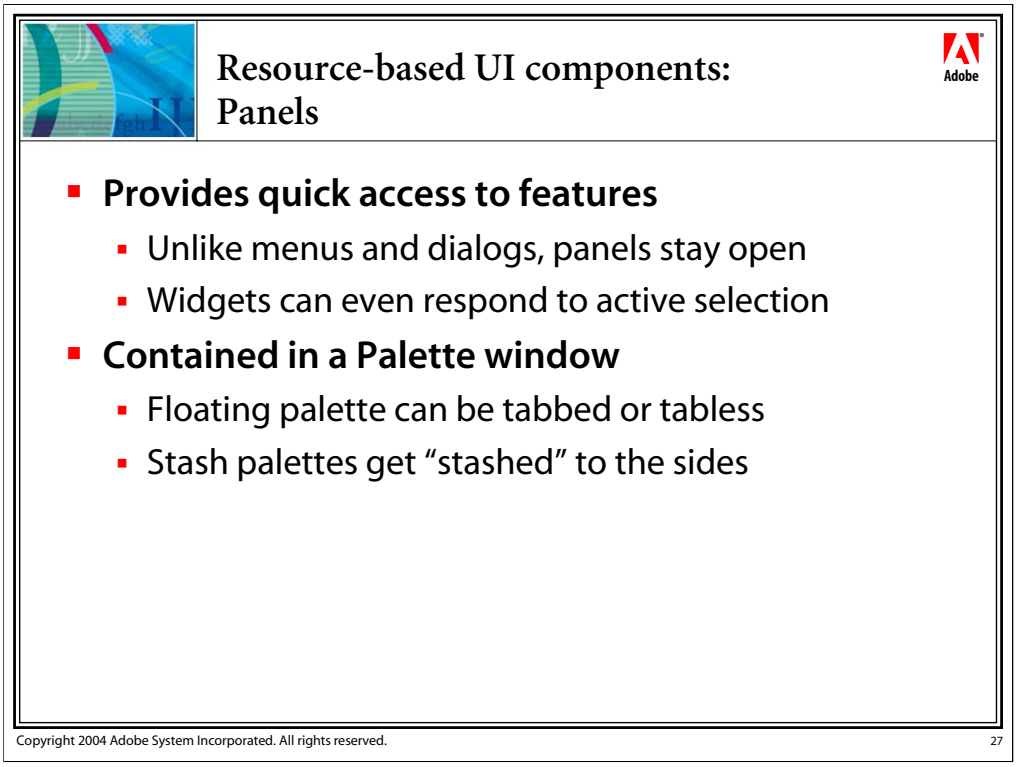
```
Class {
 kYangPanelBoss,
 kPrimaryResourcePanelWidgetBoss, {
  IID_IPANELCREATOR, kYangPanelCreatorImpl,
  IID_IK2SERVICEPROVIDER, kDialogPanelServiceImpl, } },
```

In your IPanelCreator implementation, you only need to implement the GetPanelRsrcID() method, where you declare the resource ID for the IDList that specifies the service and panel ID for this child dialog.

6. Define IDList and IDListPair resources to bind a selectable dialog service ID to the panel resource ID and plug-in ID. This is to help the selectable dialog service find the panel by it's resource ID.

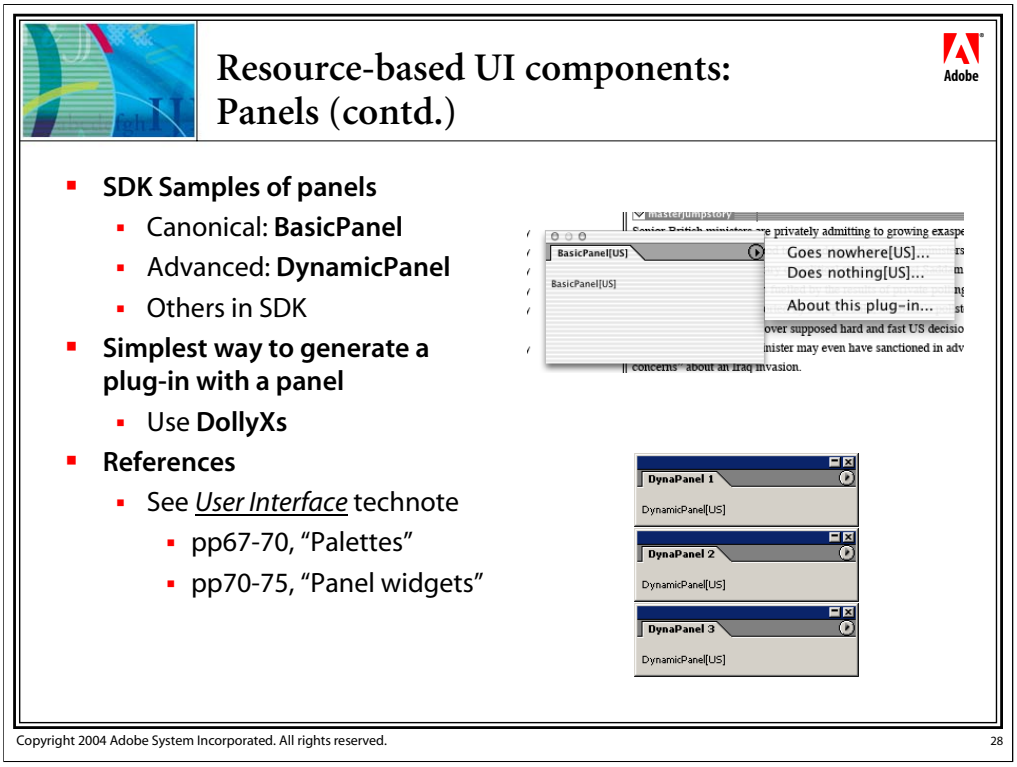
That concludes the process of implementing a child dialog for a selectable and tab selectable dialog.

The code to open the child dialog is actually in the selectable dialog service, which uses IDs specified in the parent dialog creator implementation. (As you recall, we created an IDList resource when implementing the selectable dialog parent.)


Resource-based UI components:
Panels
Adobe
Provides quick access to features
Unlike menus and dialogs, panels stay open
Widgets can even respond to active selection
Contained in a Palette window
Floating palette can be tabbed or tabless
Stash palettes get "stashed" to the sides
Copyright 2004 Adobe System Incorporated. All rights reserved.
27

A panel is another type of widget that can be a parent to other widgets. Panels provide quick access to features in a plug-in, as they stay open (as opposed to menus and dialogs), and you can set up an ActiveSelectionObserver so that the widgets on a panel respond to the active selection changes (either the selection itself or some attribute associated with the selection).

Panels are contained in a palette window. Floating palettes can be either tabbed or tabless (although tabbed palettes are more common). Non-floating palettes, or stash palettes, are palettes that get stashed to the side of the application window.

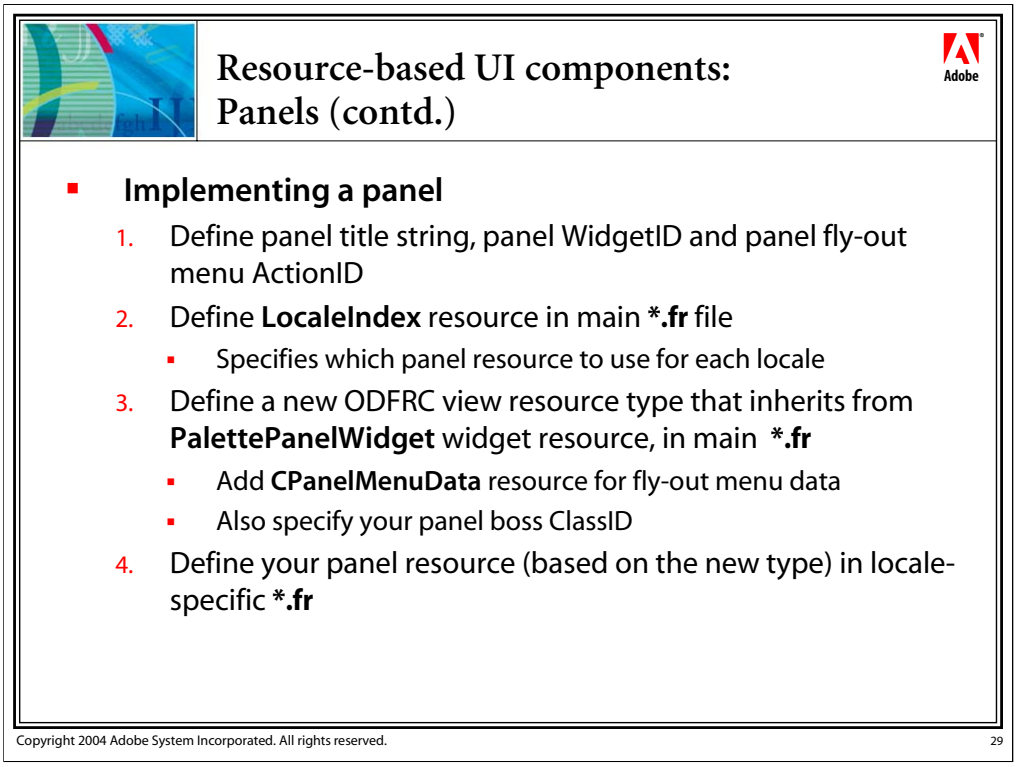


The canonical example of a panel is the BasicPanel plug-in. You can create panels dynamically (e.g. the Book and Library Asset panels are dynamically created), as shown in the DynamicPanel plug-in. There are several other SDK samples that show various types of panels and widgets that you can put on them. Look for the PalettePanelWidget resource in *.fr files.

The simplest way to generate a plug-in with a panel is by generating one with **DollyXs**. Please refer to the *Porting Guide* technote ({SDK}/docs/guides/portingguide.pdf) for details on how to use **DollyXs**.

However, if you want to implement a panel from scratch, the following slides provide the details.

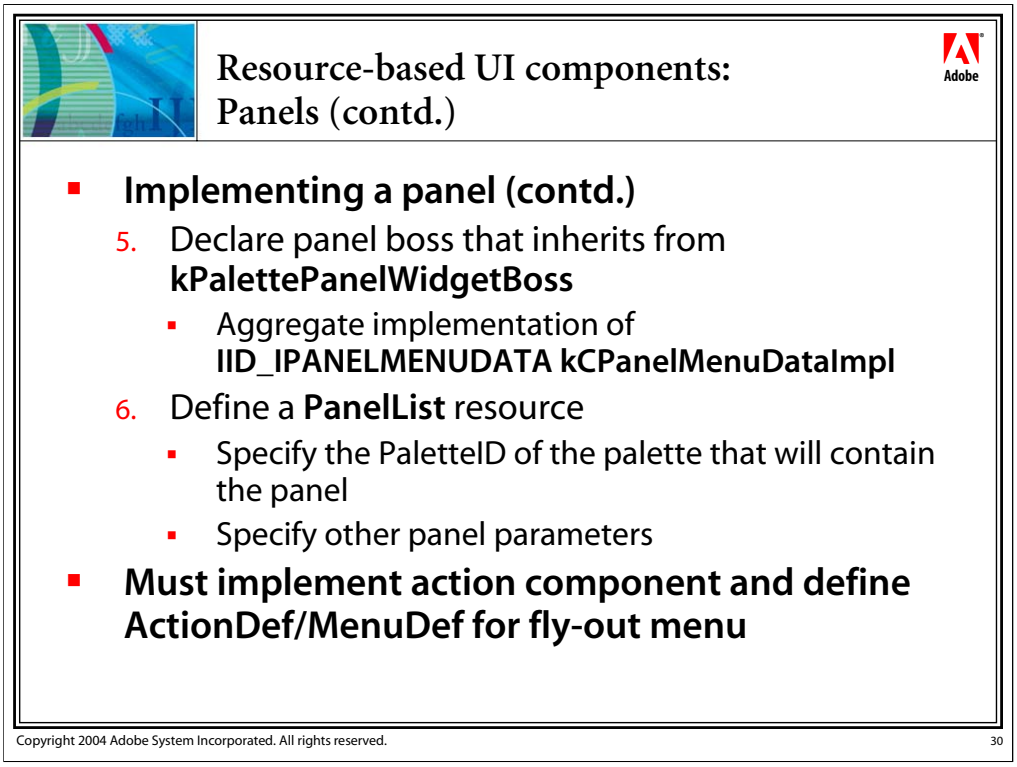
For general information on panels, please refer to the *User Interface* technote.


Resource-based UI components:
Panels (contd.)
Adobe
Implementing a panel
1. Define panel title string, panel WidgetID and panel fly-out menu ActionID
2. Define LocaleIndex resource in main *.fr file
Specifies which panel resource to use for each locale
3. Define a new ODFRC view resource type that inherits from PalettePanelWidget widget resource, in main *.fr
Add CPanelMenuData resource for fly-out menu data
Also specify your panel boss ClassID
4. Define your panel resource (based on the new type) in locale-specific *.fr
Copyright 2004 Adobe System Incorporated. All rights reserved.
29

[Presenter may skip this slide] The steps to implement a panel is as follows:

1. Define a panel title string and declare a panel widget ID. The string is created in the same way as defining a localizable string, since panels are generally specified in a locale-specific way. The WidgetID is declared in the kWidgetIDSpace using DECLARE_PMID.
2. Define a LocaleIndex resource. This resource is a mapping between locales and different panel resources, which describe the visual specification of a panel.
3. Define a new ODFRC view resource type that inherits from the PalettePanelWidget widget resource. You would specify this once in your plug-in's main *.fr file. This is also where you also specify the actual boss class that provides the implementation behind the panel widget resource. This statement looks something like this (sample from BasicPanel):
```
type BscPnlPanelWidget(kViewRsrcType) : PalettePanelWidget(ClassID =
kBscPnlPanelWidgetBoss) {
  CPanelMenuData;
};
```
4. Define your panel resource based on the new panel resource type that you just defined. You would do this in your locale-specific *.fr file. The panel resource is a container for other widgets so within this resource, you would specify other child resources. We will discuss these widgets in a later slide.

(Continued on next slide)


Resource-based UI components:
Panels (contd.)
Adobe
Implementing a panel (contd.)
5. Declare panel boss that inherits from kPalettePanelWidgetBoss
Aggregate implementation of IID_IPANELMENUDATA kCPanelMenuDataImpl
6. Define a PanelList resource
Specify the PaletteID of the palette that will contain the panel
Specify other panel parameters
Must implement action component and define ActionDef/MenuDef for fly-out menu
Copyright 2004 Adobe System Incorporated. All rights reserved.
30

[Presenter may skip this slide]

5. Declare a panel boss that inherits from kPalettePanelWidgetBoss. At the least, you need to aggregate an implementation of IPanelMenuData, but you don't need to write any code for this – you can just use IID_IPANELMENUDATA kCPanelMenuDataImpl. (If you add widgets to your panel, you might need to add an observer in this boss.)

6. Define a PanelList resource that registers a panel as part of a specific palette, and specifies other panel parameters, such as:

•Panel resource ID

•Plug-in ID of the plug-in that owns the panel

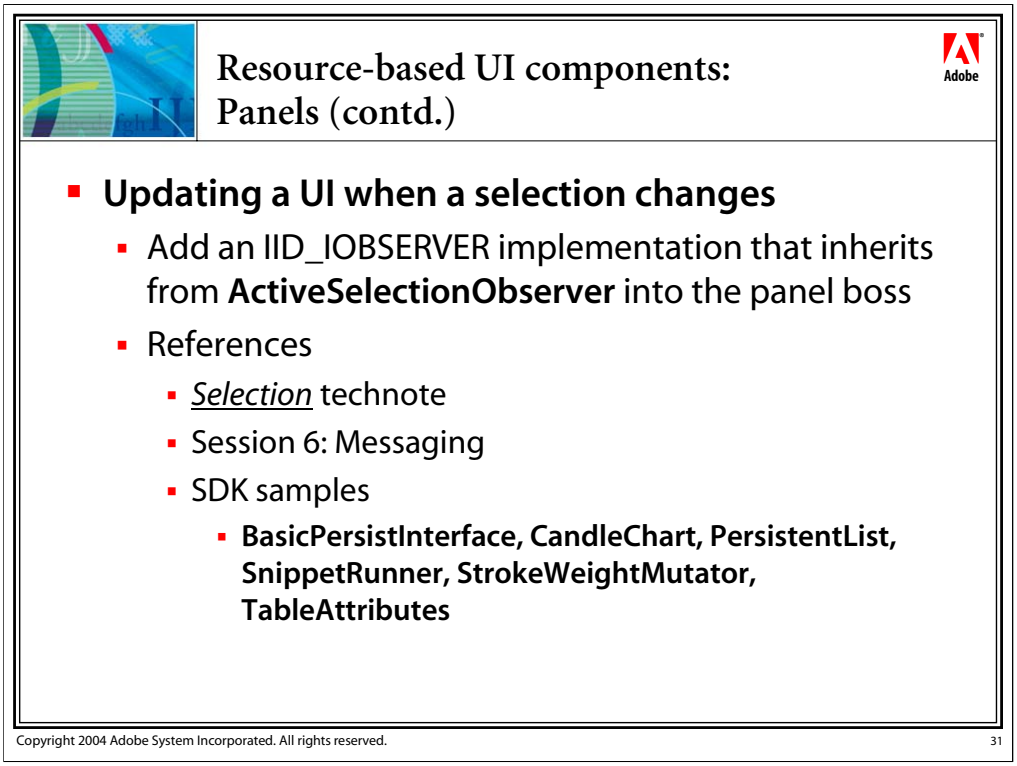
•ZOrder in the palette

•ActionID of the fly-out menu

•Title string key of panel

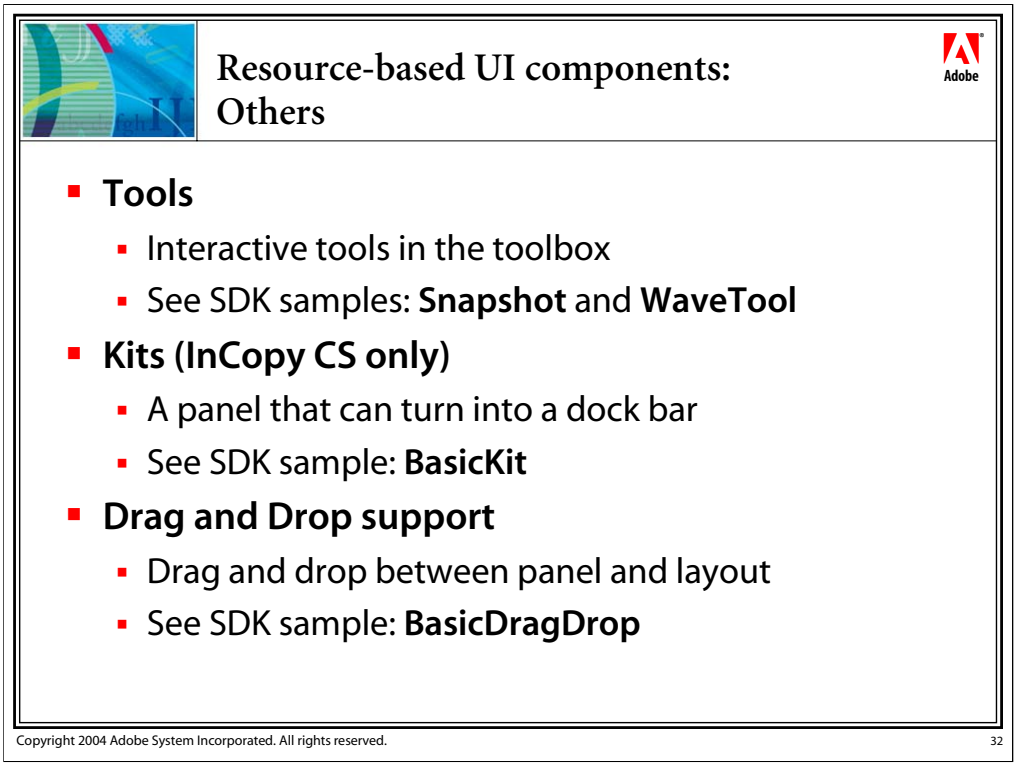
•etc.

Since there is a menu involved, you'll need to implement a menu, by following steps outlined earlier. The menu path string key can be anything since the PanelList resource knows of the ActionID associated with the fly-out menu.

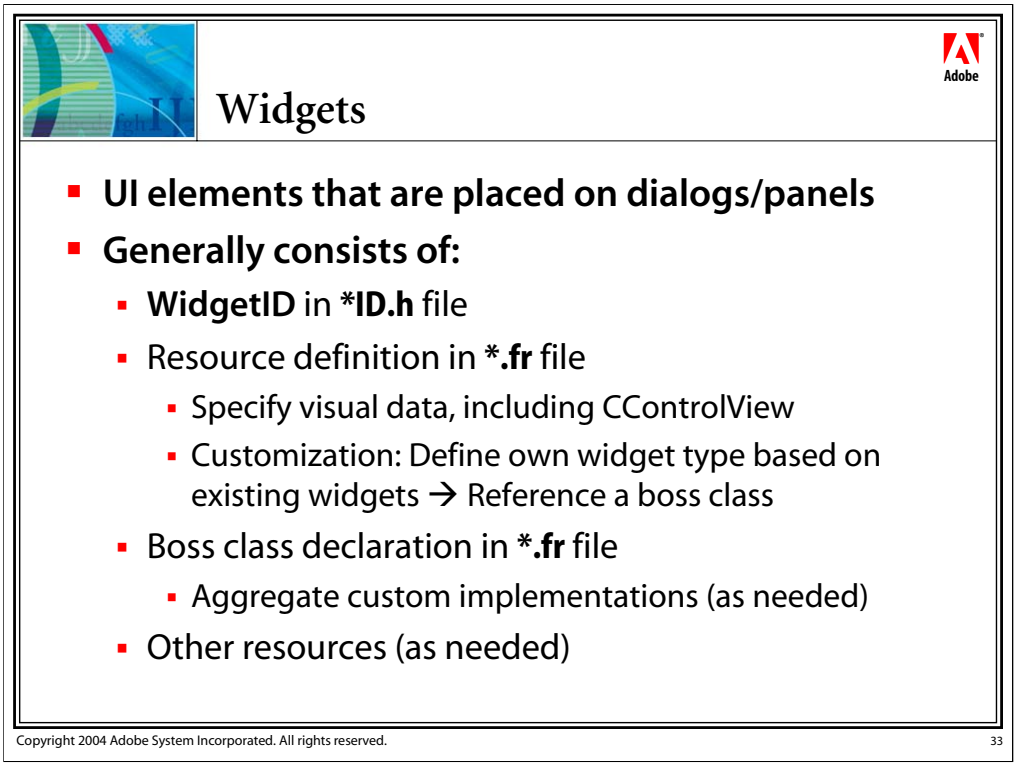

Resource-based UI components:
Panels (contd.)
Adobe
Updating a UI when a selection changes
Add an IID_IOBSERVER implementation that inherits from ActiveSelectionObserver into the panel boss
References
Selection technote
Session 6: Messaging
SDK samples
BasicPersistInterface, CandleChart, PersistentList, SnippetRunner, StrokeWeightMutator, TableAttributes
Copyright 2004 Adobe System Incorporated. All rights reserved.
31

You can update a UI widget when a selection changes. This is done by aggregating an observer implementation based on ActiveSelectionObserver in the panel boss.

You can refer to the references on the slide for more details.


Resource-based UI components:
Others
Tools
Interactive tools in the toolbox
See SDK samples: Snapshot and WaveTool
Kits (InCopy CS only)
A panel that can turn into a dock bar
See SDK sample: BasicKit
Drag and Drop support
Drag and drop between panel and layout
See SDK sample: BasicDragDrop
Copyright 2004 Adobe System Incorporated. All rights reserved.
32

This slide shows other types of UI components, simple descriptions, and SDK samples that show you how to implement them.


Widgets
Adobe
UI elements that are placed on dialogs/panels
Generally consists of:
WidgetID in *ID.h file
Resource definition in *.fr file
Specify visual data, including CControlView
Customization: Define own widget type based on existing widgets → Reference a boss class
Boss class declaration in *.fr file
Aggregate custom implementations (as needed)
Other resources (as needed)
Copyright 2004 Adobe System Incorporated. All rights reserved.
33

We have discussed that dialogs and panels are containers of widgets, or user interface elements. Let's discuss what widgets are about.

When you put a widget on a dialog or panel, you are required to do the following:

•Declare a WidgetID your plug-in's *ID.h file. The ID would be declared in the kWidgetIDSpace using the DECLARE_PMID macro.

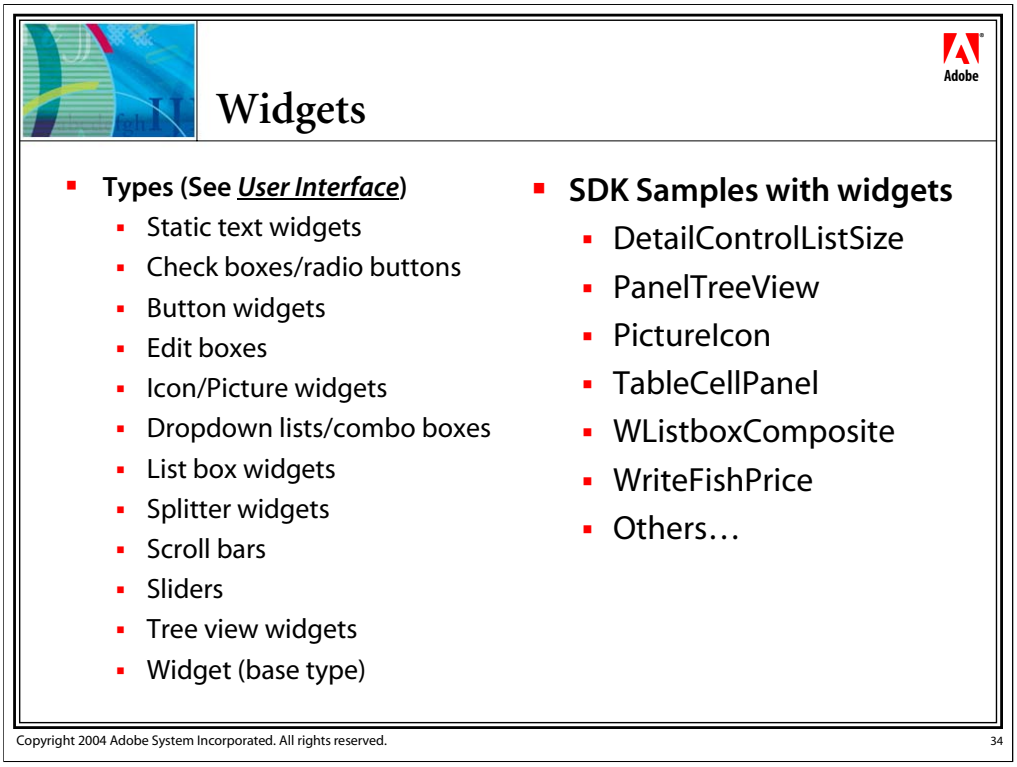
•Define a resource in your plug-in's *.fr files.

- •This is where you specify various parameters of a widget. Most widgets contain a ControlView, which is defines the visual representation of the widget. The ODFRC type is CControlView, and the interface that corresponds to this is IControlView.
- •If you want to customize the behavior of a widget, you would define your own widget type based on existing widget types (that are declared in *.fh files). When you define a widget type, you specify the boss class that provides the implementation for the widget, and this is usually your own. This way, you can associate desired widget behavior with your widget type.
- •You may even decide to customize your widget by adding extra elements. For instance, you can make a widget have an interchangeable font by adding the UIFontSpec widget element. (Your widget boss class would also need an implementation for this element, in this case, IUIFontSpec.)

•If you have defined your own widget type, you need to declare a boss class in your plug-in's *.fr file. This widget boss class would inherit from the boss class that corresponds to the base widget type you inherited when you defined your own widget type. It is in this boss class that you aggregate implementations to customize your widget's behavior.

- •If your widget needs no customization, then you can observe them from the boss of the "container widget" (dialog or panel boss), and skip the step of declaring a boss class for the widget itself.

•Some widgets require further resources to be defined.

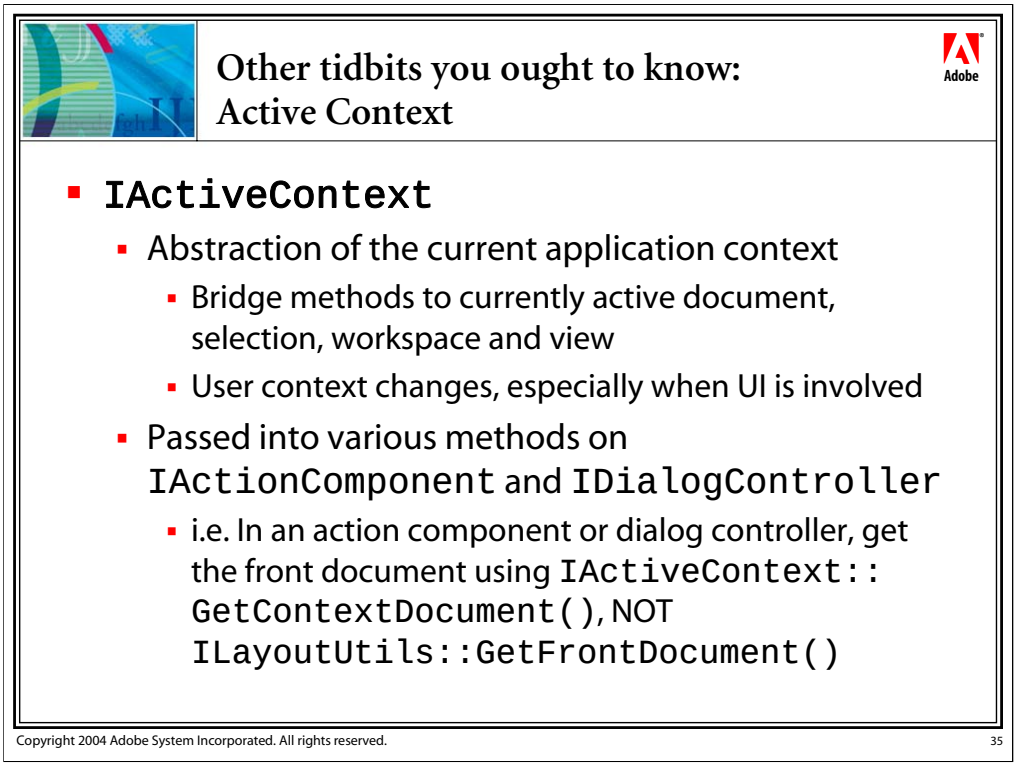


Widgets are one of the most interesting areas of the InDesign/InCopy API, as there are so many different kinds available in the API, and yet, you are not limited by these predefined types. You can extend widgets in your own ways, or define completely custom widgets by creating a widget type based on the base widget, "Widget". The widgets that are explained in detail in the *User Interface* technote are shown on the slide.

There are many other SDK samples that showcase the use of specific widgets. These range from simple examples (WriteFishPrice) to elaborate examples (PanelTreeView, TablePanelCell).

NOTE: The only SDK sample of a truly custom widget is in the TablePanelCell plug-in, which defines a TableCellWidget that is based on the base "Widget". The boss, kTableCellWidgetBoss, inherits from kBaseWidgetBoss.

Please refer to the plug-in specific documentation (HTML-based reference) for more information.


Other tidbits you ought to know:
Active Context
Adobe
IActiveContext
Abstraction of the current application context
Bridge methods to currently active document, selection, workspace and view
User context changes, especially when UI is involved
Passed into various methods on IActionComponent and IDialogController
i.e. In an action component or dialog controller, get the front document using IActiveContext::GetContextDocument(), NOT ILayoutUtils::GetFrontDocument()
Copyright 2004 Adobe System Incorporated. All rights reserved.
35

Here are some tidbits you ought to know when implementing a user interface.

The notion of an "active context", an abstraction of the current application context, is very important. This is represented by the IActiveContext interface, and gives you some idea as to what the current document, selection, workspace, and view are. This is important to know in cases where your context changes when a dialog is introduced.

IActiveContext is passed from the user interface subsystem into various methods on IActionComponent and IDialogController, which are interfaces you encounter when implementing menus/actions and dialogs. If you want to obtain the current document, selection, workspace, or view from within an implementation of IActionComponent or IDialogController, you must obtain it using the bridge methods on IActiveContext:

•virtual IDocument* GetContextDocument() const;

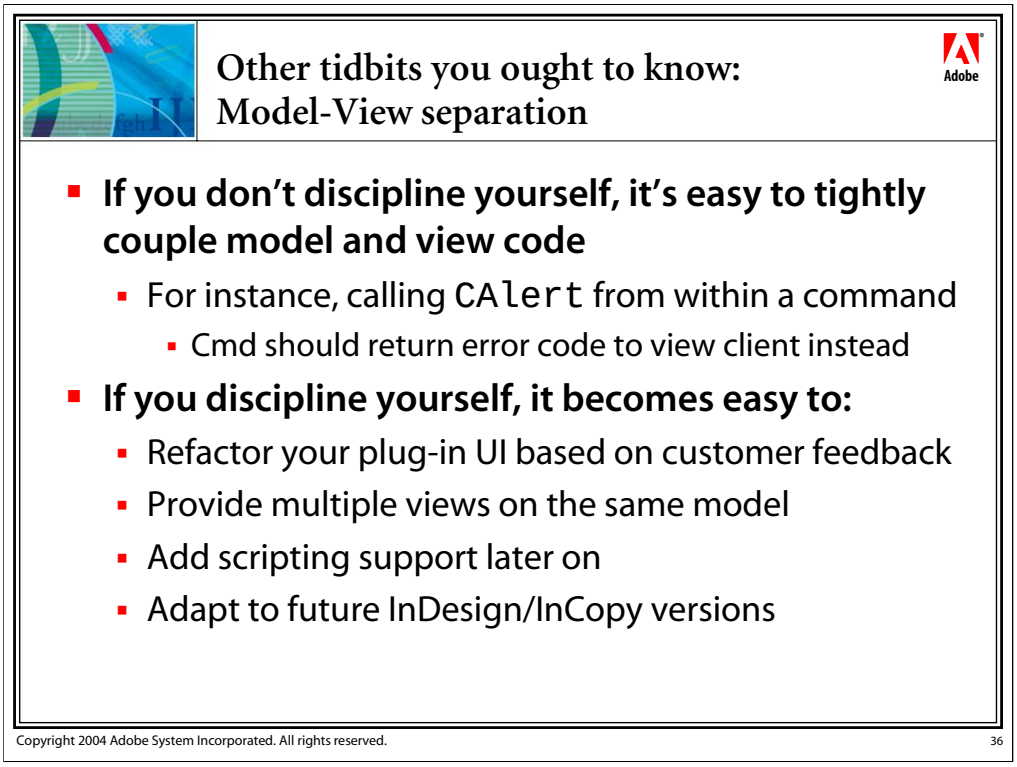
•virtual ISelectionManager* GetContextSelection() const;

•virtual IControlView* GetContextView() const;

•virtual IWorkspace* GetContextWorkspace() const;

(Can you determine which bosses they bridge to?)

In an action component or dialog controller implementation, do NOT use any other methods to obtain the current context object, such as ILayoutUtils::GetFrontDocument(), to obtain the current document.


Other tidbits you ought to know:
Model-View separation
Adobe
If you don't discipline yourself, it's easy to tightly couple model and view code
For instance, calling CAlert from within a command
Cmd should return error code to view client instead
If you discipline yourself, it becomes easy to:
Refactor your plug-in UI based on customer feedback
Provide multiple views on the same model
Add scripting support later on
Adapt to future InDesign/InCopy versions
Copyright 2004 Adobe System Incorporated. All rights reserved.
36

Another tidbit you ought to consider when implementing a user interface, is to design your plug-in(s) so that the model and view components are clearly decoupled. It is very easy to mix in view components within model code, for instance, using of CAlert from within a command. The suggested way to deal with this is for the command to return a status code, and if the client (presumably a view component) of the command sees an error, it can then issue an alert.

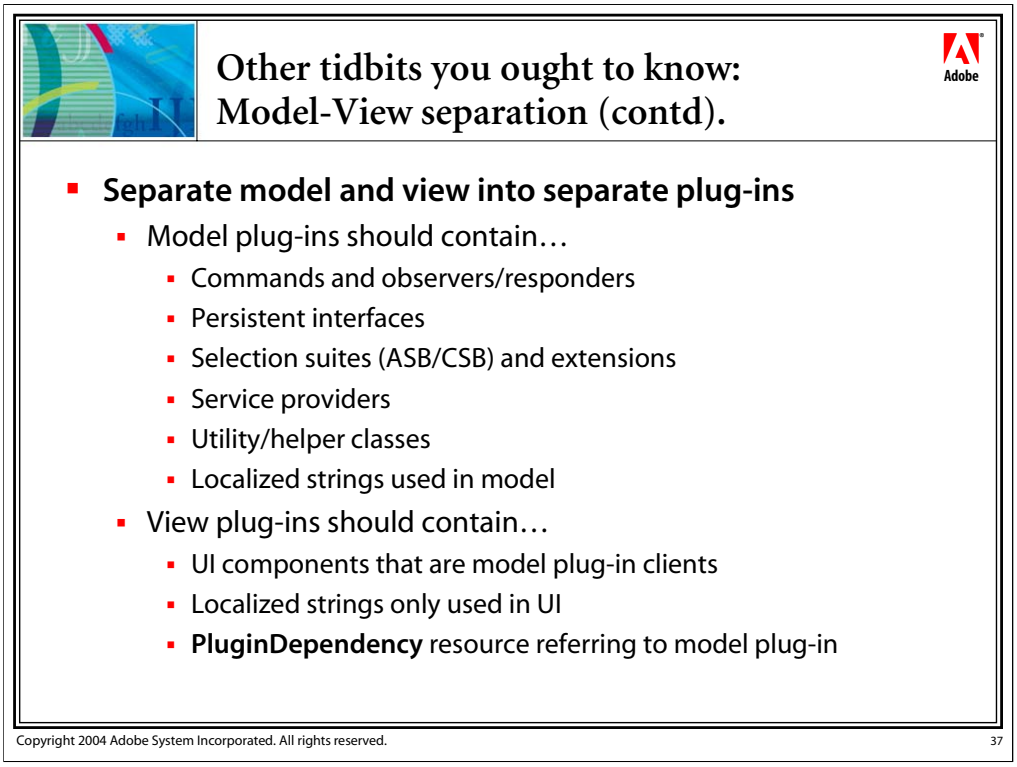
If you discipline yourself from the beginning, it becomes very easy to:

•Refactor your plug-in UI based on customer feedback. If a customer didn't like the way a UI worked, you can completely redevelop your UI-only plug-in in the next version, while the separate model data plug-in is left untouched.

•Provide multiple views on the same model. An example: Character panel and Character dockbar in InDesign CS.

•Add scripting support later on. Scripting support can be thought of as another "user interface" that is driven by scripts instead of mouse clicks and key presses. By having a model-only set of components, you can easily wrap the scripting components around it without impacting any existing user interface components.

•Adapt to future versions of InDesign and InCopy versions. The user interface specifications do change somewhat from version to version. By decoupling your view components from your model components, it becomes easier to follow these specification changes.


Other tidbits you ought to know:
Model-View separation (contd).
Adobe
Separate model and view into separate plug-ins
Model plug-ins should contain…
Commands and observers/responders
Persistent interfaces
Selection suites (ASB/CSB) and extensions
Service providers
Utility/helper classes
Localized strings used in model
View plug-ins should contain…
UI components that are model plug-in clients
Localized strings only used in UI
PluginDependency resource referring to model plug-in
Copyright 2004 Adobe System Incorporated. All rights reserved.
37

Here are some guidelines you can use to separate model and view plug-ins.

Model plug-ins should contain implementations of the following (but not limited to):

•Custom commands

•Observers/responders that watch changes to model data (not including user interface observers)

•Persistent interfaces (AddIns into kWorkspaceBoss/kDocWorkspaceBoss, other persistent bosses)

•Selection suites (ASB/CSB) and extensions that manipulate selection items

•Service providers

•Utility/helper classes (e.g. interfaces added into kUtilsBoss)

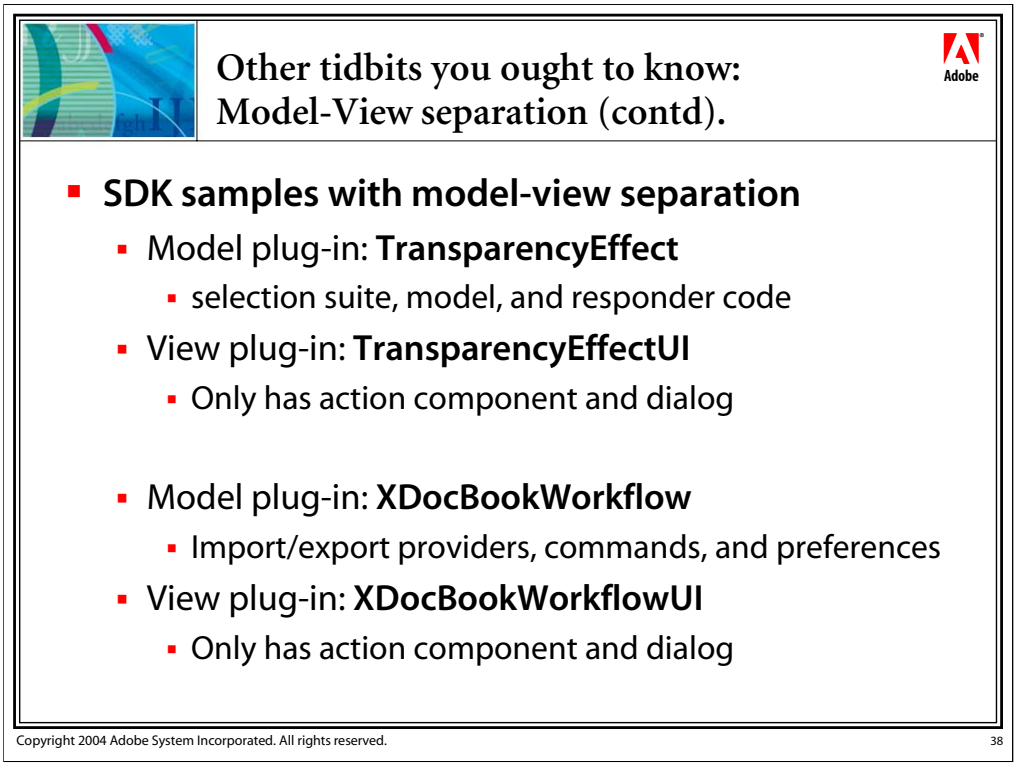
•Localized strings used in any model component

View plug-ins should contain the following (but not limited to):

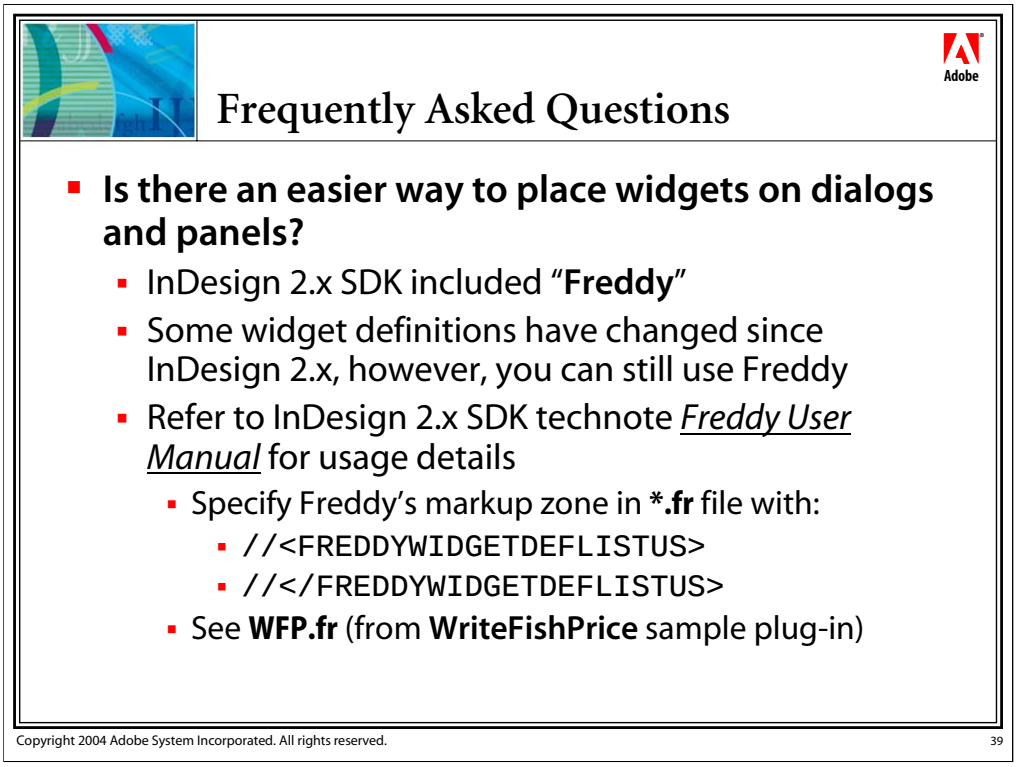
•Any UI component that is a client of the model plug-ins, such as abstract selection observers, selection suite client code, etc.

•Localized strings used in UI components (such as dialogs, panels, menus, and any widget)

•A PluginDependency resource that indicates a dependence on the model plug-in.


Other tidbits you ought to know:
Model-View separation (contd).
Adobe
SDK samples with model-view separation
Model plug-in: TransparencyEffect
selection suite, model, and responder code
View plug-in: TransparencyEffectUI
Only has action component and dialog
Model plug-in: XDocBookWorkflow
Import/export providers, commands, and preferences
View plug-in: XDocBookWorkflowUI
Only has action component and dialog
Copyright 2004 Adobe System Incorporated. All rights reserved.
38

See the slide for SDK sample plug-ins that are to be used in tandem. These sets of plug-ins are offer model-view separation, as described on the slide. See the HTML-based reference documentation for more details of these plug-ins.



Here are some frequently asked questions related to the implementation of plug-in user interfaces.

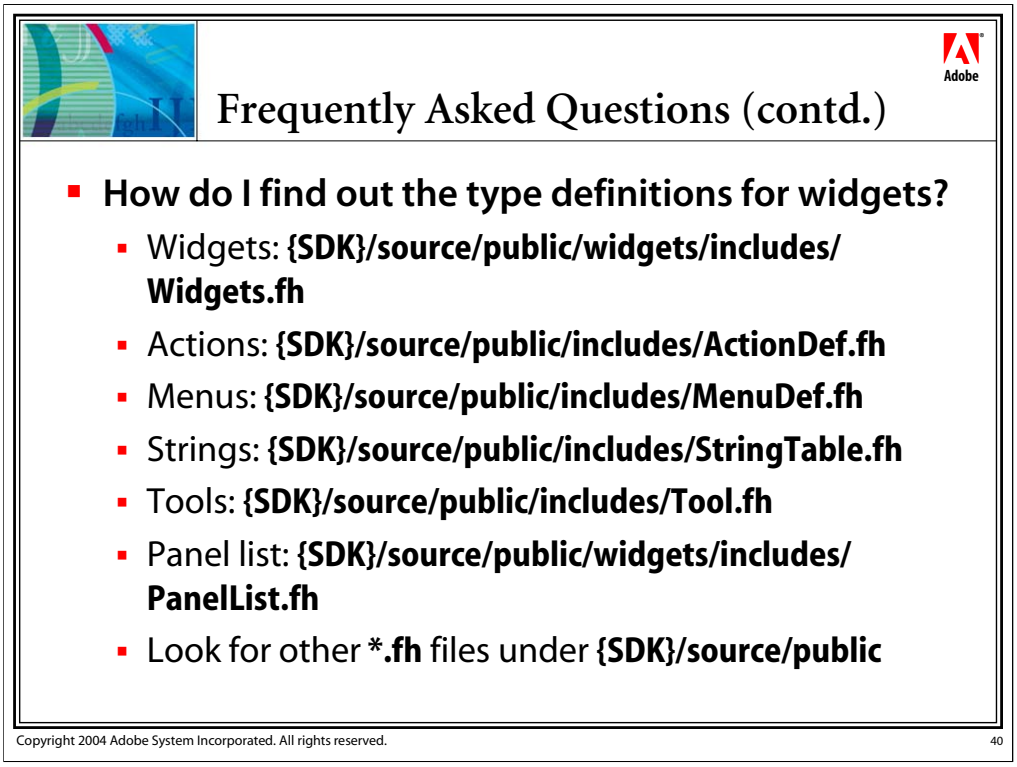
It would be nice if there was an interactive way to design user interfaces, like the Resource Editor in Visual Studio. The InDesign CS/InCopy CS Combined SDK doesn't contain such a tool, however, the InDesign 2.x SDK included a tool called "Freddy", which was a nickname based on the acronym for "Framework Resource Editor".

Freddy is a Java-based interactive UI designer, and can write common widget resources for dialogs and panels directly into an *.fr file that has the "Freddy markup zone" markers (C++ style comments with XML-like tags). Some of the widget definitions have changed since InDesign 2.x, so after editing with Freddy, you may have to hand-edit what Freddy inserted into the *.fr file (by comparing it with the types in *.fh files), nonetheless, Freddy might provide a shortcut for the UI design process.

You still have to do the necessary C++ coding yourself.

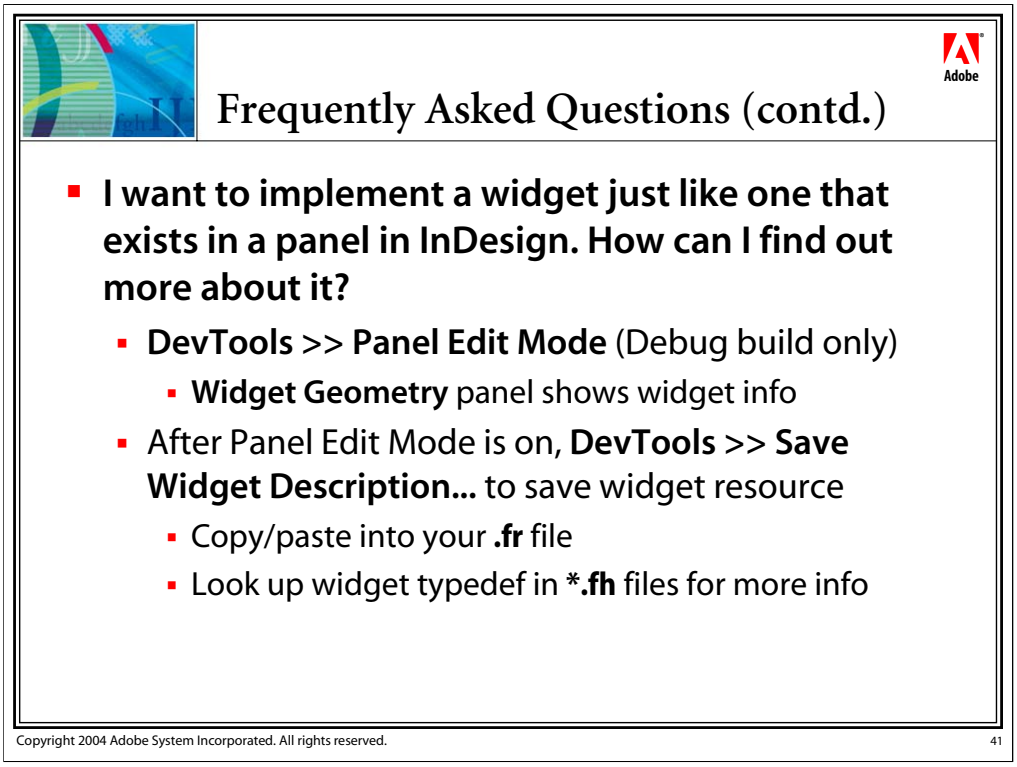
If you have the InDesign 2.x SDK installed on your machine, you can try this out. To try out a sample that has the Freddy markup zone, refer to WFP.fr from the WriteFishPrice sample plug-in. To find out how to use Freddy, refer to the InDesign 2.x SDK technote *Freddy User Manual*.

You can download the InDesign 2.x SDKs (2.0, 2.0.1, 2.0.2) from http://partners.adobe.com/asn/indesign/sdk.jsp.



Whether or not use decide to use Freddy for UI design, you will need to find out the various widget types. This slide shows the SDK ODFRC header files that contain the type definitions.

Most widgets are typedef'ed in Widgets.fh.


Frequently Asked Questions (contd.)
Adobe
I want to implement a widget just like one that exists in a panel in InDesign. How can I find out more about it?
DevTools >> Panel Edit Mode (Debug build only)
Widget Geometry panel shows widget info
After Panel Edit Mode is on, DevTools >> Save Widget Description... to save widget resource
Copy/paste into your .fr file
Look up widget typedef in *.fh files for more info
Copyright 2004 Adobe System Incorporated. All rights reserved.
41

Sometimes, the best way to learn is by imitation. By using the debug build of InDesign CS and InCopy CS, you can utilize the Panel Edit Mode. First, open a panel with the widget of interest. You then enable Panel Edit Mode from the DevTools >> Panel Edit Mode menu in the debug build, which will open the Widget Geometry panel. Then click on the widget of interest, and look at the Widget Geometry panel for detailed information about the widget, such as:

•Widget boss class (You can look up the widget boss in HTML-based reference docs to see what implementations are available, what you can reuse, and what will be inherited if you inherit a related widget type.)

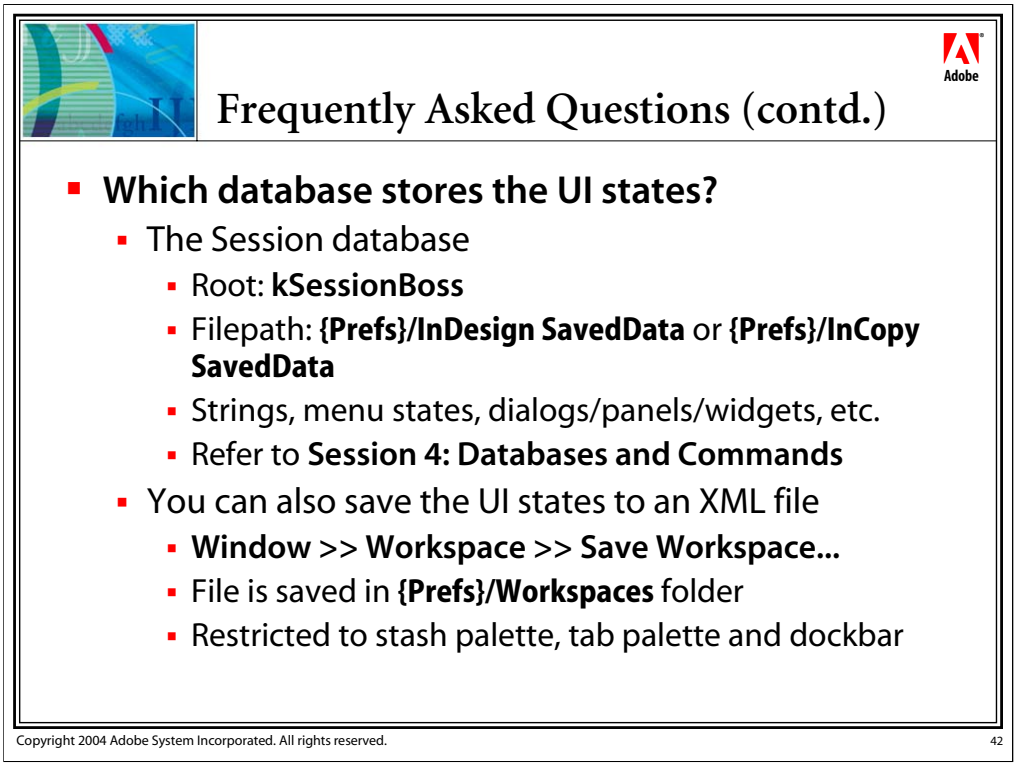
•Widget ID

•Widget parent

•etc.

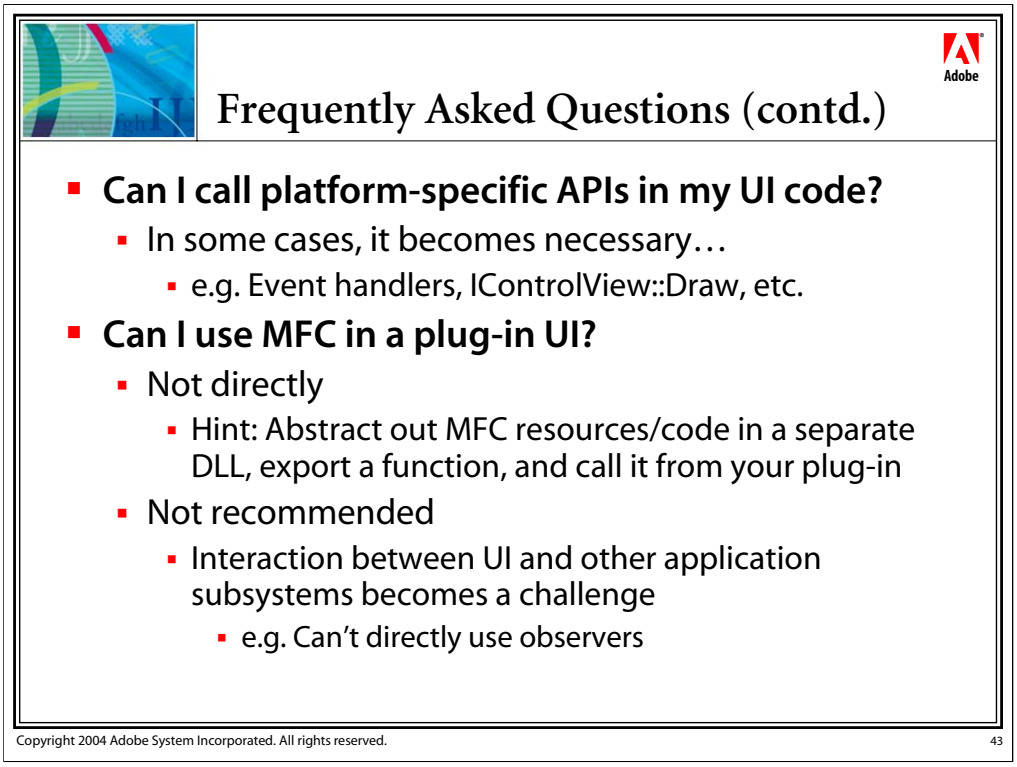
You can then go back to the DevTools menu for some more options, such as saving the widget description to a file. You can then copy and paste the saved widget resource into your own *.fr file. You can find out more information about the typedef in *.fh files in the SDK.

The catch is, you still have to do necessary C++ coding yourself, just like with Freddy-generated widget definitions.


Frequently Asked Questions (contd.)
Which database stores the UI states?
The Session database
Root: kSessionBoss
Filepath: {Prefs}/InDesign SavedData or {Prefs}/InCopy SavedData
Strings, menu states, dialogs/panels/widgets, etc.
Refer to Session 4: Databases and Commands
You can also save the UI states to an XML file
Window >> Workspace >> Save Workspace...
File is saved in {Prefs}/Workspaces folder
Restricted to stash palette, tab palette and dockbar
Copyright 2004 Adobe System Incorporated. All rights reserved.
42

You may remember this from Session 4: Databases and Commands. The Session database (root boss: kSessionBoss, filepath: "{Product} Saved Data" where {Product} is either InDesign or InCopy) is the database that stores UI states, such as strings, menu states, dialog/panels/widgets, etc.

Starting with InDesign CS/InCopy CS, there is a product feature where you can save some of the UI states to an XML file, so you can restore the state later. The procedures are as stated in the slide. The UIs that are included in the XML file are restricted to panels housed in stash palettes, tab palettes, and the dock bar. Developers may use this to setup a particular set of custom panel configurations for users.


Frequently Asked Questions (contd.)
Can I call platform-specific APIs in my UI code?
In some cases, it becomes necessary…
e.g. Event handlers, IControlView::Draw, etc.
Can I use MFC in a plug-in UI?
Not directly
Hint: Abstract out MFC resources/code in a separate DLL, export a function, and call it from your plug-in
Not recommended
Interaction between UI and other application subsystems becomes a challenge
e.g. Can't directly use observers
Copyright 2004 Adobe System Incorporated. All rights reserved.
43

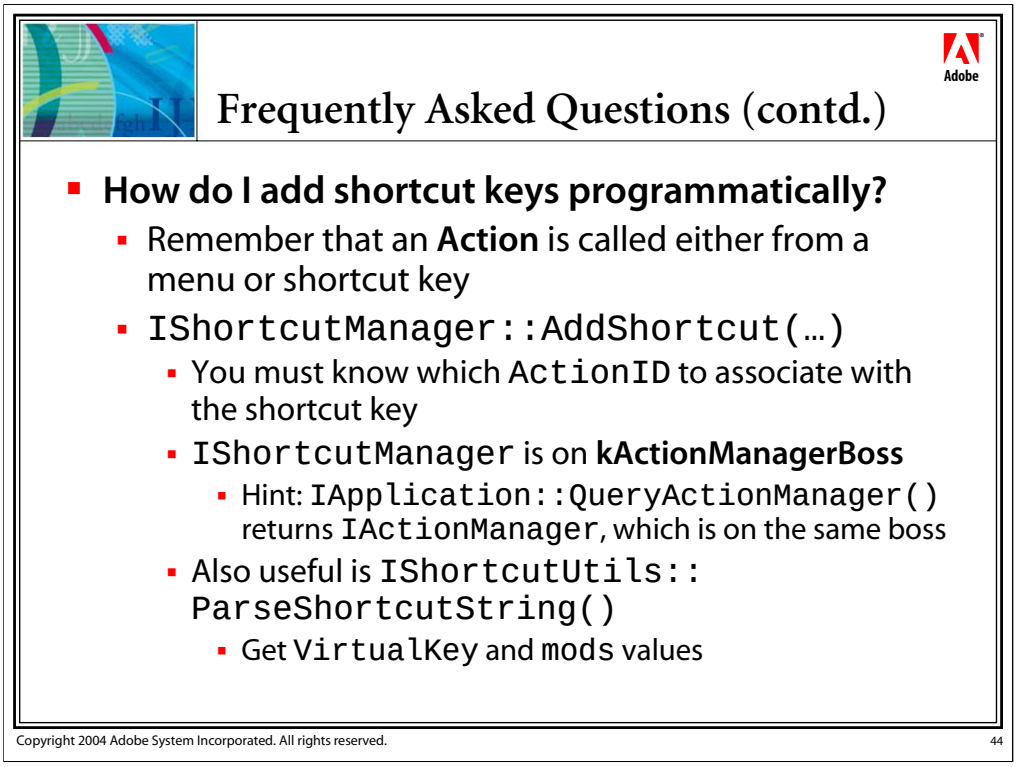
Here are some questions related to platform-specific implementations.

Sometimes, it becomes necessary to call platform specific APIs whenever you have to interface with the operating system directly to handle an event or draw on a widget. The InDesign/InCopy C++ API does have a cross-platform drawing API called IGraphicsPort (which has methods similar to PostScript), however, there are cases where this isn't sufficient. In those cases, you will need to call platform specific APIs. When you have to do that, you can either use #ifdef MACINTOSH or #ifdef WINDOWS macros to tell the C++ preprocessor to compile only code that is relevant to the platform, or you can abstract out entire class implementations. For instance, you might have a MMyControlView.cpp (Macintosh) and a WMyControlView.cpp (Windows), and both implementations the same Implementation ID in the CREATE_PMINTERFACE macro. However, by compiling the MMyControlView.cpp only in your CodeWarrior project and the WMyControlView.cpp only in your Visual C++ project, you can provide platform specific implementations of a control view without cluttering up your code with #ifdefs. The only drawback is that you have to prepare two separate implementations. (Of course another approach is to provide a Mac-only or Windows-only plug-in, however, we do recommend that plug-ins be provided equally to users of both platforms.)

A related question: Some developers have asked if they could use the UI frameworks that they are already comfortable with, for example, Microsoft Foundation Classes (MFC) or PowerPlant for CodeWarrior.

If you abstract out the platform-specific constructs into a completely separate component (DLL, shared library), this is possible. The slide gives a hint for this approach.

The official answer is that this is not recommended. For instance, there isn't a direct way for the InDesign/InCopy application's notification mechanisms to notify these platform-specific constructs, and vice versa. You might have to provide your own adapter API in your plug-in (remember: a plug-in is a DLL or shared library) to make notification between the two components work properly. Also, the use of platform-specific APIs is generally not supported by Adobe Developer Support, so you will have a difficult time getting support through paid developer cases.


Frequently Asked Questions (contd.)
How do I add shortcut keys programmatically?
Remember that an Action is called either from a menu or shortcut key
IShortcutManager::AddShortcut(...)
You must know which ActionID to associate with the shortcut key
IShortcutManager is on kActionManagerBoss
Hint: IApplication::QueryActionManager() returns IActionManager, which is on the same boss
Also useful is IShortcutUtils::ParseShortcutString()
Get VirtualKey and mods values
Copyright 2004 Adobe System Incorporated. All rights reserved.
44

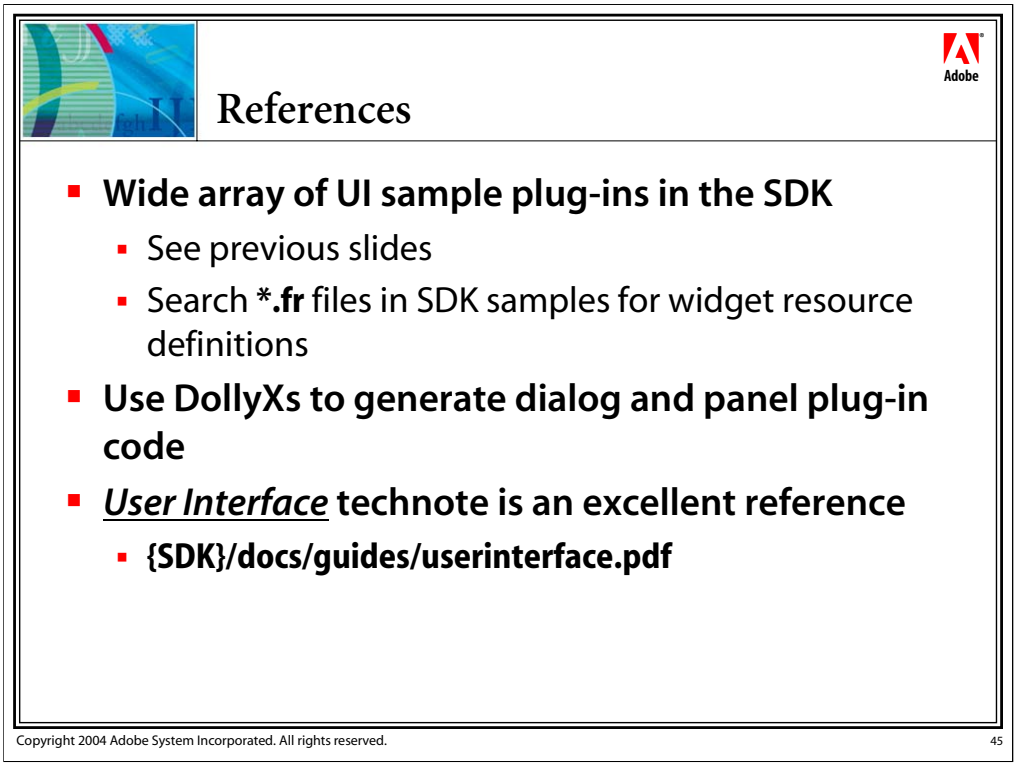
Shortcut keys can be added from the Edit >> Keyboard shortcuts menu, but how do you add them programmatically?

IShortcutManager has a method to do just that.

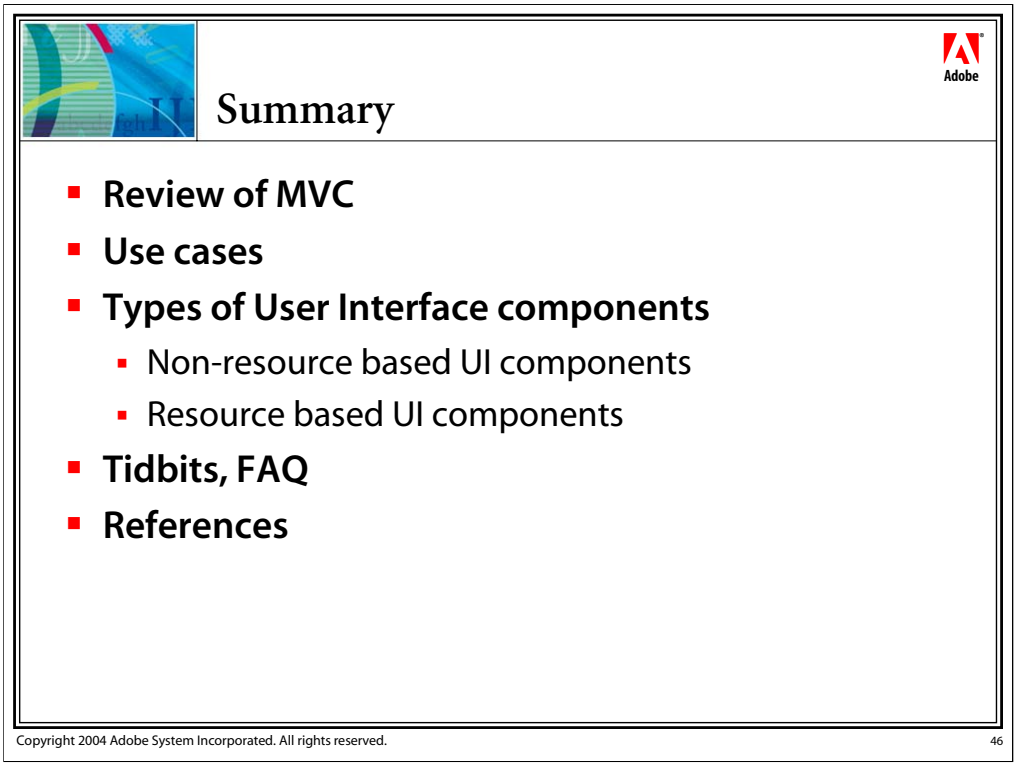
See slide for hints on how to query IShortcutManager. (Do you remember how to query for IApplication?)

IShortcutUtils (kUtilsBoss) also has some useful methods.

Now, you can choose an Action Area for the shortcut on the Keyboard Shortcuts dialog. This is the action area that you define in the ActionDef resource, and it groups actions by name.


References
Wide array of UI sample plug-ins in the SDK
See previous slides
Search *.fr files in SDK samples for widget resource definitions
Use DollyXs to generate dialog and panel plug-in code
User Interface technote is an excellent reference
{SDK}/docs/guides/userinterface.pdf
Copyright 2004 Adobe System Incorporated. All rights reserved.
45

There are many references you can rely on to learn more about user interface implementation.

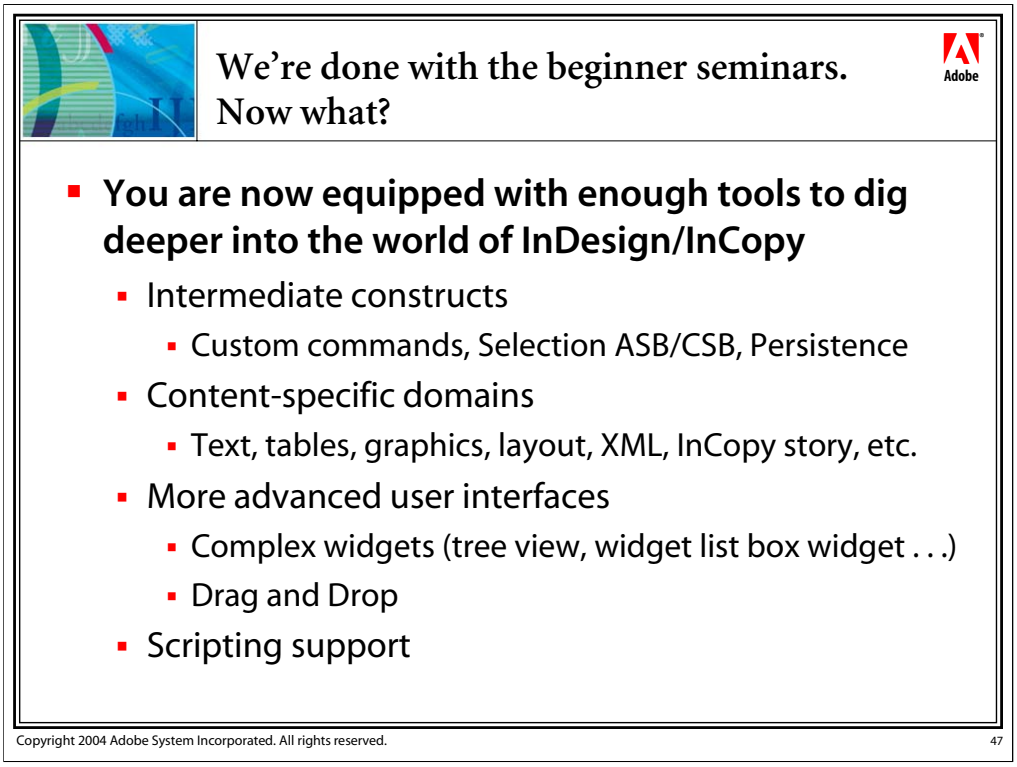


In this session, we reviewed the MVC paradigm and discussed where the UI fit into the picture (the view, but also part controller). We then discussed scenarios (use cases) in which UI implementation is useful. We then discussed the different types of UI elements you can use and implement. Some don't require you to define ODFRC resources, some do.

Many design and implementation tips were shared through tidbits you ought to know, and FAQs.

Then we looked at references for further exploration.

This is a good time for Q&A, time permitting.

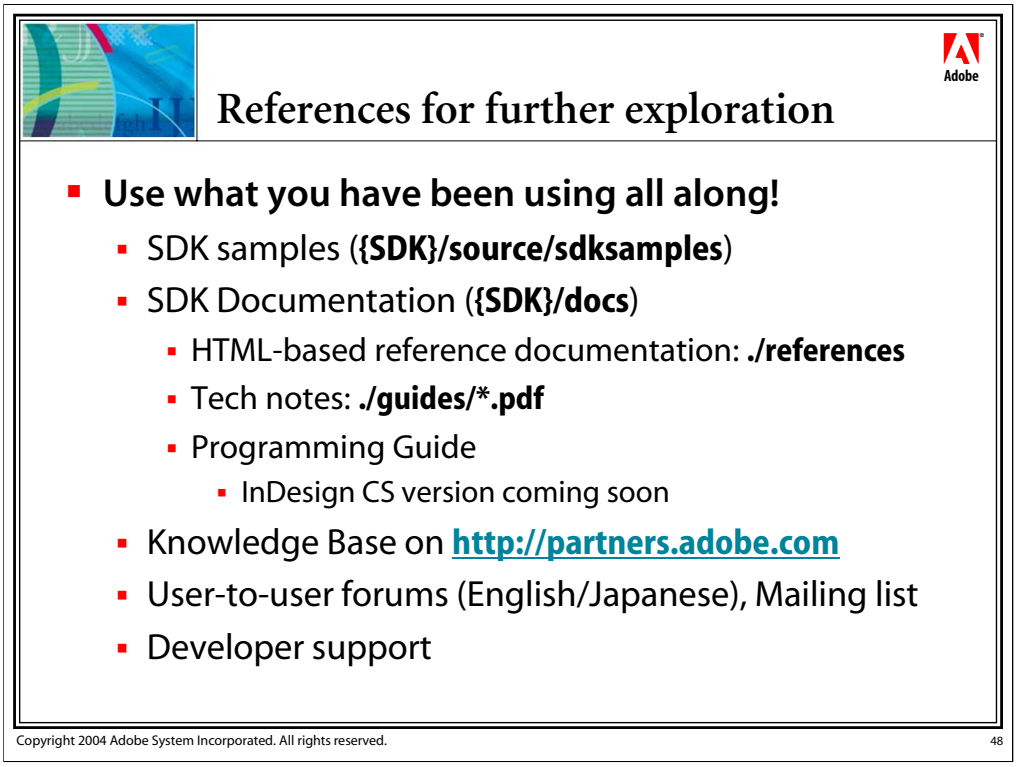

We're done with the beginner seminars. Now what?
Adobe
You are now equipped with enough tools to dig deeper into the world of InDesign/InCopy
Intermediate constructs
Custom commands, Selection ASB/CSB, Persistence
Content-specific domains
Text, tables, graphics, layout, XML, InCopy story, etc.
More advanced user interfaces
Complex widgets (tree view, widget list box widget . . .)
Drag and Drop
Scripting support
Copyright 2004 Adobe System Incorporated. All rights reserved.
47

This is the last of the beginner series of InDesign CS/InCopy CS plug-in development training sessions.

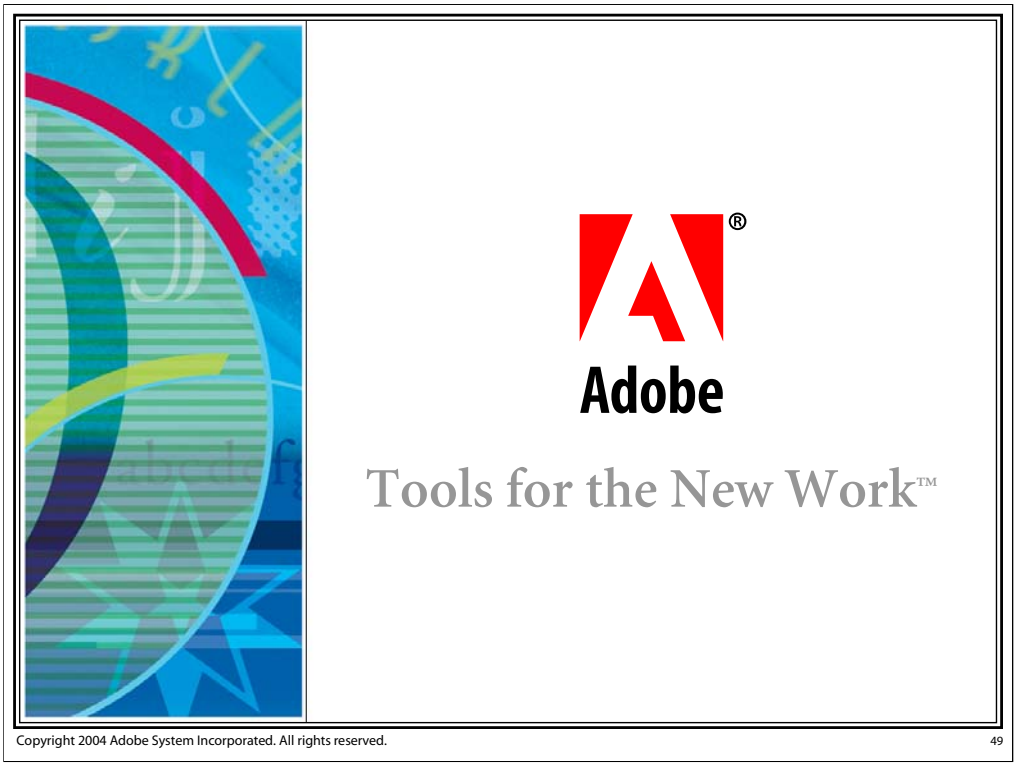
After you have established a firm foundation with these 8 session, you are now equipped with sufficient fundamental knowledge to explore other domains in the InDesign/InCopy application. Namely:

- Intermediate constructs
  - Custom commands: How to implement a custom command
  - Selection ASB/CSB and extensions: How to implement an ASB/CSB and Selection extension
  - Persistence: How to add persistent interfaces/bosses, reference conversion, and dealing with data format changes from version to version
- Content-specific domains
  - Text: how to access and manipulate the text model, text attributes, text layout, etc.
  - Tables: how to access and manipulate the table model, table attributes, table layout, etc.
  - Graphics: how to access and manipulate graphic attributes, placing images/PDFs, color/gradients/swatches, transparency, color management, etc.
  - Layout: how to access and manipulate page items
  - XML: how to access and manipulate the logical structure of a document (as opposed to document hierarchy), mapping styles to tags, managing tags, etc.
  - InCopy story: importing/exporting, working with InCopy features
- Advanced user interfaces
  - Complex widgets, such as widget list box widget, tree view widget, etc.
  - Drag and Drop support
- Adding scripting support for your plug-ins

All of these topics are advanced topics that we didn't cover in these series, however, you can use the references (listed on the next slide) to explore more on these topics.


References for further exploration
Use what you have been using all along!
SDK samples ({SDK}/source/sdksamples)
SDK Documentation ({SDK}/docs)
HTML-based reference documentation: ./references
Tech notes: ./guides/*.pdf
Programming Guide
InDesign CS version coming soon
Knowledge Base on http://partners.adobe.com
User-to-user forums (English/Japanese), Mailing list
Developer support
Copyright 2004 Adobe System Incorporated. All rights reserved.
48

Just a reminder, there are plenty of other resources in the SDK and outside that you can refer to for further exploration of the world of InDesign CS/InCopy CS plug-in development.



All of us at Adobe wish you success with your plug-in development!

Thank you very much for attending the beginner series of Adobe InDesign CS/InCopy CS Plug-in development training sessions.